Записки за упражнения по ДАА

Тодор Дуков

1 август 2024 г.

Съдържание

Предисловие **iii**

1 Въведение в алгоритмите и асимптотичния анализ **1**
1.1 Що е то алгоритъм? . . . 1
1.2 Какво означава добро решение? . . . 2
1.3 Как мерим времето и паметта? . . . 2
1.4 Основни дефиниции . . . 3
1.5 Полезни свойства . . . 6
1.6 Задачи . . . 7

2 Анализ на сложността на алгоритми **9**
2.1 Как анализираме един алгоритъм по сложност? . . . 9
2.2 Предимствата и недостатъците на този вид анализ . . . 10
2.3 Сложност по време на някои итеративни алгоритми . . . 11
2.4 Защо са ни рекурентни уравнения? . . . 13
2.5 Начини за намиране на асимптотиката на рекурентни уравнения . 14
2.6 Задачи . . . 16

3 Коректност на алгоритми **19**
3.1 Какво имаме предвид под коректност? . . . 19
3.2 Едно *"ново"* понятие – инвариант . . . 19
3.3 Инвариантите в действие . . . 20
3.4 С инвариантите трябва да се внимава . . . 21
3.5 Подход при задачи с вече даден алгоритъм . . . 22
3.6 Примери за рекурсивни алгоритми . . . 25
3.7 Трик за бързо пресмятане на членове на някои рекурентни редици 26
3.8 Още примери . . . 28
3.9 Задачи . . . 30

4 Сортиране и неговите приложения **36**
4.1 Каква задача решаваме? . . . 36
4.2 Сортиране чрез сливане . . . 36
4.3 Бързо сортиране . . . 39
4.4 Два алгоритъма . . . 42
4.5 Задачи . . . 45

5 Алгоритми върху графи **47**
5.1 Защо изобщо се занимаваме с графи? . . . 47
5.2 Как представяме графите в паметта? . . . 48
5.3 Код на алгоритмите за обхождане на графи . . . 49
5.4 Най-къси пътища в тегловен граф . . . 51
5.5 Структурата Union-find/Disjoint-union . . . 51
5.6 Алгоритъмът на Крускал за намиране на МПД . . . 53
5.7 Задачи . . . 53

6 Динамично програмиране **56**
6.1 Какво е динамично програмиране? . . . 56
6.2 Прости примери за динамично програмиране . . . 56
6.3 Динамично програмиране за решаване на комбинаторни задачи . . . 58
6.4 Два интересни примера . . . 60
6.5 Задачи . . . 64

7 Долни граници **67**
7.1 Какво са долни граници? . . . 67
7.2 Техники за изследване на долни граници . . . 68
7.3 Техниките в действие . . . 69
7.4 Задачи . . . 71

8 Алгоритмична неподатливост **74**
8.1 Класове на сложност **P** и **NP** . . . 74
8.2 Няколко важни задачи за класа **NP** . . . 75
8.3 $\mathbf{P} = \mathbf{NP}$ или $\mathbf{P} \neq \mathbf{NP}$? Колко пък да е трудно? . . . 79
8.4 *“Основната”* **NP**-пълна задача . . . 80
8.5 Класически **NP**-пълни задачи . . . 81
8.6 Задачи . . . 84

Предисловие

Тези записки представляват кратко въведение в света на алгоритмите. Тук се опитах да събера основните концепции и техники, които ще са ви нужни за създаването и анализирането на алгоритми.
Искам обаче да натъртя, че тези записки НЕ СА завършени и най-вероятно никога няма да бъдат. Първо, липсват решения и доказателства на много от разгледаните въпроси. Също така е много вероятно да съдържат множество грешки – граматични, правописни, пунктуационни и логически. Така че, моля ви, не ги използвайте като основен източник на информация. Те са създадени най-вече за да подредя собствените си мисли и евентуално да ви послужат като допълнителен материал по време на вашето обучение.
Въпреки че тези записки не са перфектни, се надявам те да бъдат полезни и интересни за някои от вас. Може би ще намерите в тях полезна информация, която ще ви помогне да разберете по-добре материала и да се справите успешно с предизвикателствата в изучаването на алгоритми.

Благодаря ви, че отделихте време да разгледате тези записки.
Надявам се, че ще ви бъдат полезни.
Тед

Глава 1

Въведение в алгоритмите и асимптотичния анализ

1.1 Що е то алгоритъм?

Алгоритмите се срещат навсякъде около нас:

- рецептите са алгоритми за готвене;
- сутрешното приготвяне;
- придвижването от точка A до точка B;
- търсенето на книга в библиотеката.

Въпреки това е трудно да се даде формална дефиниция на това какво точно е алгоритъм. На ниво интуиция, човек може да си мисли, че това просто е някакъв последователен списък от стъпки/инструкции, които човек/машина трябва да изпълни. Други начини човек да си мисли за алгоритмите, са:

- програми – обикновено така се реализират алгоритми;
- машини на Тюринг, крайни (стекови) автомати или формални граматики;
- частично рекурсивни функции.

Един програмист в ежедневието си постоянно пише алгоритми за да решава различни задачи/проблеми. Една задача може да се решава по много начини, някои по-добри от други. Добрият програмист, освен че ще намери решение на проблема, той ще намери най-доброто решение (или поне достатъчно добро за неговите цели).

1.2 Какво означава добро решение?

Хубаво е човек да се води по следните (неизчерпателни) критерии:

- решението трябва да е коректно – ако алгоритъмът работи само през 50% от времето, най-вероятно можем да се справим по-добре;
- решението трябва да е бързо – ако алгоритъмът ще завърши работа след като всички звезди са измрели, то той практически не ни върши работа;
- решението трябва да заема малко памет – ако алгоритъмът по време на своята работа се нуждае от повече памет, колкото компютърът може да предостави, за нас този алгоритъм е безполезен;
- решението трябва да е просто – това е може би най-маловажният критерии от тези, но въпреки това е хубаво когато човек може, да пише чист и разбираем код, който лесно се разширява.

За да можем да сравняваме алгоритми в зависимост от това колко големи ресурси (време и памет) използват, трябва първо да можем да "измерваме" тези ресурси.

1.3 Как мерим времето и паметта?

Когато пишем алгоритми, имаме няколко базови инструкции (за които предварително сме се уговорили), които ще наричаме **атомарни инструкции**. Тяхното извикване ще отнеме една единица време. **Време за изпълнение** ще наричаме броят на извикванията на атомарните инструкции по време на изпълнение на програмата. Също така числата и символите ще бъдат нашите **атомарни типове данни**, и ще заемат една единица памет. **Паметта**, която една програма заема, ще наричаме максималния брой на единици от атомарни типове данни по време на изпълнение, без да броим входните данни. Обикновено времето и паметта зависят от размера на подадените входни данни. Това означава, че можем да си мислим за времето и паметта като функции на размера на входа. Подходът, който ще изберем, е да сравняваме функциите за време/памет на различните алгоритми асимптотично. Интересуваме се не толкова от конкретните стойности, а от поведението им, когато размерът на входа клони към безкрайност.

1.4 Основни дефиниции

Множеството от функции, които ще анализираме, е

$$\mathcal{F} = \{f \mid f : \mathbb{R}^{\geq 0} \to \mathbb{R} \ \& \ (\exists n_0 > 0)(\forall n \geq n_0)(f(n) > 0)\}.$$

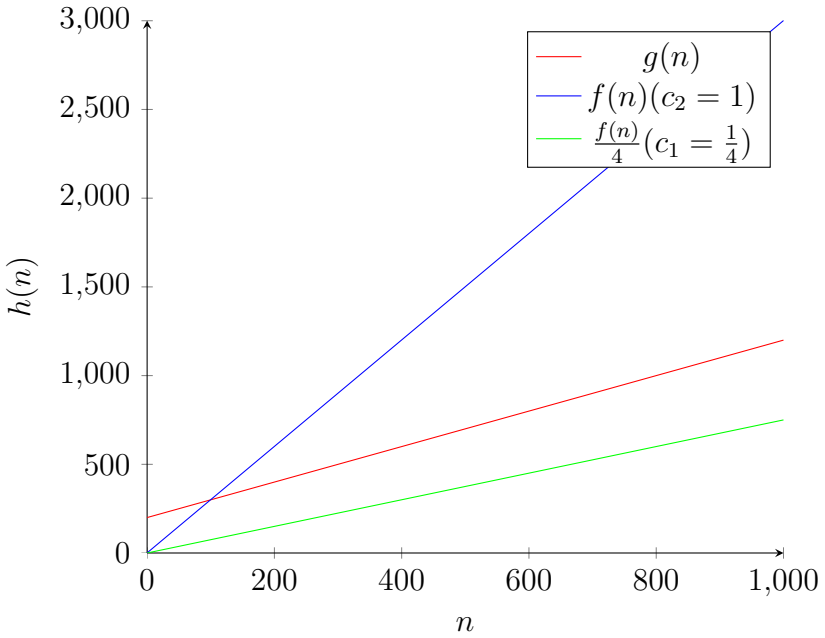
Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\Theta(f) = \{g \in \mathcal{F} \ | (\exists c_1 > 0)(\exists c_2 > 0) \\ (\exists n_0 \geq 0)(\forall n \geq n_0)(c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n))\}.$$

Може да тълкуваме $\Theta(f)$ като:

"множеството от функциите, които растат със скоростта на f".*

Нека вземем за пример $f(n) = 3n + 1$ и $g(n) = n + 200$:



На картинката се вижда как от един момент нататък, функцията g остава "заключена" между $c_1 \cdot f$ и $c_2 \cdot f$. Точно заради това $g \in \Theta(f)$.

Забележка. Вместо да пишем $g \in \Theta(f)$, ще пишем $g = \Theta(f)$ или $g \asymp f$.

*точност до константен множител и константно събираемо

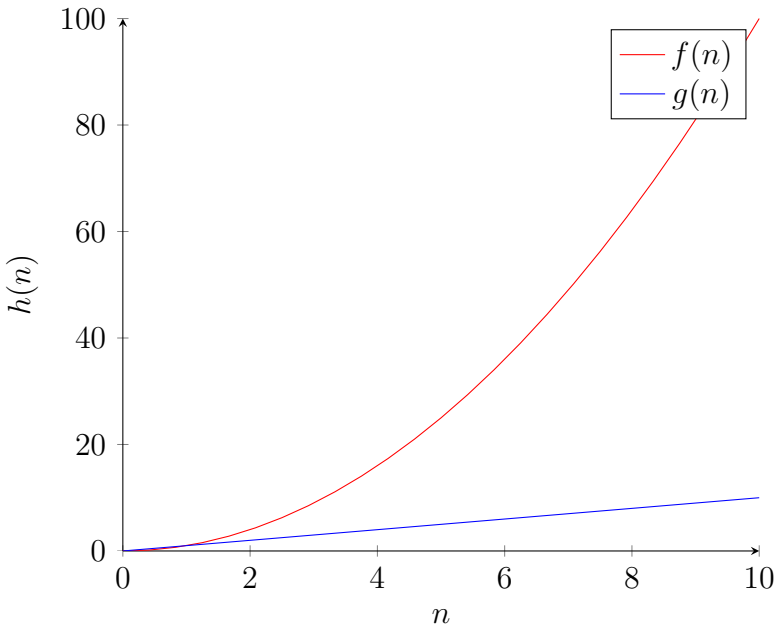
Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$O(f) = \{g \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(g(n) \leq c \cdot f(n))\}.$$

Може да тълкуваме $O(f)$ като:

"множеството от функциите, които не растат по-бързо от f".*

Тук заслабваме условията от $\Theta(f)$ като искаме само горната граница.
За пример човек може да вземе $f(n) = n^2$ и $g(n) = n$:



Забележка. Вместо да пишем $g \in O(f)$, ще пишем $g = O(f)$ или $g \preceq f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$o(f) = \{g \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(g(n) < c \cdot f(n))\}.$$

Може да тълкуваме $o(f)$ като:

"множеството от функциите, които растат по-бавно от f".*

Разликата между $O(f)$ и $o(f)$ е строгото неравенство и универсалният квантор в началото. Лесно се вижда, че $o(f) \subseteq O(f)$. Тук изключваме функциите от същия порядък.

Забележка. Вместо да пишем $g \in o(f)$, ще пишем $g = o(f)$ или $g \prec f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\Omega(f) = \{g \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(c \cdot f(n) \leq g(n))\}.$$

Може да тълкуваме $\Omega(f)$ като:

"множеството от функциите, които не растат по-бавно от f".*

Това е дуалното множество на $O(f)$.

Забележка. Вместо да пишем $g \in \Omega(f)$, ще пишем $g = \Omega(f)$ или $g \succeq f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\omega(f) = \{g \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(c \cdot f(n) < g(n))\}.$$

Може да тълкуваме $\omega(f)$ като:

"множеството от функциите, които растат по-бързо от f".*

Това е дуалното множество на $o(f)$.

Забележка. Вместо да пишем $g \in \omega(f)$, ще пишем $g = \omega(f)$ или $g \succ f$.

Внимание. Не всички функции от $\mathcal{F}$ са сравними по релациите $\prec, \preceq$ или $\asymp$. За пример човек може да вземе функциите $f(n) = n$ и $g(n) = n^{1+\sin(n)}$. Лесно се вижда, че функцията $g(n)$ "плава" между $n^0 = 1$ и n^2 т.е. няма нито как да расте по-бързо, нито как да расте по-бавно.

Въпреки това, тези релации са сравнително хубави.

Твърдение 1.4.1. *Следните свойства са в сила:*

- $\asymp$ *е релация на еквивалентност;*
- $\prec$ *и* $\succ$ *са транзитивни и антирефлексивни;*
- $\preceq$ *и* $\succeq$ *са транзитивни и рефлексивни.*

Доказателството на това твърдение оставяме за упражнение на читателя. То е една елементарна разходка из дефинициите.

1.5 Полезни свойства

Тук ще изброим няколко свойства, които много често се ползват в задачите:

- Нека $f, g \in \mathcal{F}$ и $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = l$ (тук искаме границата да съществува). Тогава:
  — ако $l = 0$, то $f \prec g$;
  — ако $l = \infty$, то $f \succ g$;
  — в останалите случаи $f \asymp g$.
- $f + g \asymp \max\{f, g\}$ за всяко $f, g \in \mathcal{F}$.
- $c \cdot f \asymp f$ за всяко $f \in F$ и $c > 0$.
- $f \asymp g \iff f^c \asymp g^c$ за всяко $f, g \in F$ и $c > 0$.
- $O(f) \cap \Omega(f) = \Theta(f)$ за всяко $f \in \mathcal{F}$.
- $o(f) \cap \omega(f) = O(f) \cap \omega(f) = o(f) \cap \Omega(f) = \varnothing$ за всяко $f \in \mathcal{F}$.
- $f \prec g \iff g \succ f$ и $f \preceq g \iff g \succeq f$ за всяко $f, g \in \mathcal{F}$.
- Нека $f, g \in \mathcal{F}$ са растящи и неограничени и $c > 1$. Тогава:
  - ако $f \prec g$, то $c^f \prec c^g$;
  - ако $\log_c(f) \prec \log_c(g)$, то $f \prec g$;
  - ако $c^f \asymp c^g$, то $f \asymp g$;
  - ако $f \asymp g$, то $\log_c(f) \asymp \log_c(g)$.
- Тъй като $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$, то $\log_a(n) \asymp \log_b(n)$ – вече ще пишем само $\log(n)$, като ще имаме предвид $\log_2(n)$.
- $n! \asymp \sqrt{n}\frac{n^n}{e^n}$ – от апроксимация на Стирлинг. Тук по-общият резултат е

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1.$$

- $\log(n!) \asymp n\log(n)$.
- $\log(n) \prec n^k \prec 2^n \prec n! \prec n^n \prec 2^{n^2}$ за всяко $k \geq 1$.

Твърдение 1.5.1. *Нека $f \in \mathcal{F}$ е неограничено растяща и диференцируема, $a > 1$ и $k, l > 0$. Тогава $(\log_a f(n))^k \prec f(n)^l$.*

Доказателство. Първо правим следната преработка:

$$\frac{f(n)^l}{(\log_a f(n))^k} = \frac{f(n)^{l \cdot \frac{k}{k}}}{(\log_a f(n))^k} \overset{t=\frac{l}{k}}{=} \frac{f(n)^{tk}}{(\log_a f(n))^k} = \left(\frac{f(n)^t}{\log_a f(n)}\right)^k.$$

След това забелязваме, че:

$$\begin{aligned}
&\lim_{n\to\infty} \frac{f(n)^t}{\log_a f(n)} \\
&= \lim_{n\to\infty} \frac{(f(n)^t)'}{(\log_a f(n))'} && // \text{ правило на L'Hôpital} \\
&= \lim_{n\to\infty} \frac{t f(n)^{t-1} f'(n)}{\frac{f'(n)}{\ln(a) f(n)}} \\
&= \lim_{n\to\infty} t \ln(a) f(n)^t && // \ f(n) \text{ е неограничено растяща и } t, \ln(a) > 0 \\
&= \infty.
\end{aligned}$$

Тъй като $\lim\limits_{n\to\infty} \frac{f(n)^t}{\log_a f(n)} = \infty$ и $k > 0$, то тогава:

$$\lim_{n\to\infty} \frac{f(n)^l}{(\log_a f(n))^k} = \lim_{n\to\infty} \left(\frac{f(n)^t}{\log_a f(n)}\right)^k = \infty.$$

Така $(\log_a f(n))^k \prec f(n)^l$. □

1.6 Задачи

Задача 1.1. Да се сравнят асимптотично следните двойки функции:

1. $f(n) = \log(\log(n))$ и $g(n) = \log(n)$;
2. $f(n) = 5n^3$ и $g(n) = n\sqrt{n^9 + n^5}$;
3. $f(n) = n5^n$ и $g(n) = n^2 3^n$;
4. $f(n) = n^n$ и $g(n) = 3^{n^2}$;
5. $f(n) = 3^{n^2}$ и $g(n) = 2^{n^3}$.

Задача 1.2. Да се докаже, че $\sum_{i=0}^{n} i^k \asymp n^{k+1}$.

Задача 1.3. Да се подредят по асимптотично нарастване следните функции:

$$f_1(n) = n^2 \qquad f_2(n) = \sqrt{n} \qquad f_3(n) = \log^2(n) \qquad f_4(n) = \sqrt{\log(n)!}$$

$$f_5(n) = \sum_{k=2}^{\log(n)} \frac{1}{k} \qquad f_6(n) = \log(\log(n)) \qquad f_7(n) = 2^{2^{\sqrt{n}}} \qquad f_8(n) = \binom{\binom{n}{3}}{2}$$

$$f_9(n) = 2^{n^2} \qquad f_{10}(n) = 3^{n\sqrt{n}} \qquad f_{11}(n) = 2^{\binom{n}{2}} \qquad f_{12}(n) = \sum_{k=1}^{n^2} \frac{1}{2^k}.$$

Задача 1.4. Да се докаже, че съществуват $f, g \in \mathcal{F}$, за които $f \asymp g$ и въпреки това $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ не съществува.

Глава 2

Анализ на сложността на алгоритми

2.1 Как анализираме един алгоритъм по сложност?

Нека започнем с един прост пример:

```
1  Find(A[1...n] ∈ array(ℤ); v ∈ ℤ):
2      for i ← 1 to n:
3          if A[i] = v: return i
4
5      return −1
```

Да кажем, че искаме да проверим броя на инструкциите, която тази функция ще изпълни, преди да приключи работата си. Точен отговор не може да се даде. В зависимост от това къде се намира v във $A[1 \dots n]$, алгоритъмът може да приключи много бързо или много бавно. Можем да дадем горна и долна граница на бързодействието.
Ако v се намира в началото, то ще сме направили само следните 4 операции:

- да инициализираме променливата i със 0;
- да проверим верността на $i < n$;
- да проверим верността на $A[i] = v$;
- да върнем i т.е. 1.

Нека сега да помислим какво ще стане в най-лошия случай (обикновено от тези ще се интересуваме) – v не участва в $A[1 \dots n]$. Тогава n пъти ще изпълним следните 3 операции:

- проверяваме верността на $i \leq n$;
- проверяваме верността на $A[i] = v$;
- увеличаваме i с 1.

Освен тези $3n$ операции, преди всичко трябва да инициализираме променливата i със 1, да се направи последната проверка на верността на $i \leq n$ (която ще ни изкара от цикъла), и да върнем -1. Общо излизат $3n + 3$ операции.
Така виждаме, че в зависимост от входните данни, алгоритъмът приключва работа за поне 4 стъпки и най-много $3n + 3$ стъпки. Такъв алгоритъм ще казваме, че има сложност по време $O(n)$. Разбира се, няма да е грешно и да кажем, че алгоритъмът има сложност по време $\Omega(1)$, но това не ни дава никаква информация, защото всеки алгоритъм има такава сложност. Също така, понеже не използваме допълнителни променливи, алгоритъмът ни има константна сложност по памет или сложност по памет $\Theta(1)$.
По-общо казано, се интересуваме от асимптотиката на $T(n)$, където $T(n)$ е броят елементарни инструкции, които алгоритъмът извиква по време на своето изпълнение, при вход с размер n в най-лошият случай.
Тук вход с големина n може да означава различни неща. Ако входът е някакъв масив или множество, то под размер ще разбираме броят на елементи. Ако пък входът е число, то под размер можем да разбираме самата стойност на числото или дължината на двоичния запис.

2.2 Предимствата и недостатъците на този вид анализ

Най-голямото предимство на асимптотичния анализ, е неговата простота. Вместо да влачим някакви константни множители и събираеми, имаме колкото се може по-проста формула, която да описва сложността на нашия алгоритъм. Това дали един алгоритъм работи със две или три стъпки по-бързо/бавно не ни интересува особено много. При много голям вход те ще работят практически еднакво. В някакъв смисъл това ни помага да виждаме по-голямата картинка. Един алгоритъм може да бъде по-бърз от друг, но от по-бърз алгоритъм до по-бърз алгоритъм има голяма разлика.

Нека вземем за пример следната таблица:

n	$\lceil \log_2(n) \rceil$	n	n^2	2^n
1	0	1	1	2
10	4	10	100	1024
100	7	100	10000	число със 31 цифри
10000	13	10000	100000000	число със 3011 цифри
1000000	20	1000000	1000000000000	число със 301030 цифри

Алгоритъм със сложността n^2 ще е по-бавен от алгоритъм със сложност n, обаче скока в бързината е много по-малък от този между 2^n и n^2.
Този подход обаче си има своите недостатъци. Нека разгледаме два алгоритъма със сложности по време съответно n и $2^{2^{2^{1024}}}$. Ние ведната ще се втурнем да кажем, че първият алгоритъм е по-лош. Той е с линейна сложност, а вторият алгоритъм има константна сложност. Обаче преди вторият алгоритъм даде отговор, всички звезди ще умрат т.е. няма да доживем да чуем този отговор. Разбира се, от някъде нататък, за много голями входни данни, първият алгоритъм наистина ще работи по-бавно, но ние никога няма да работим с толкова големи данни. Тогава на практика, първият алгоритъм е по-добър, нищо че асимптотично се води по-лош. Нас това няма да ни интересува в курса по ДАА.

2.3 Сложност по време на някои итеративни алгоритми

Нека видим сложността на алгоритъма за сортиране по метода на мехурчето:

```
1  Sort(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n − 1:
3          for j ← 1 to n − i − 1:
4              if A[j] > A[j + 1]:
5                  temp ← A[j]
6                  A[j] ← A[j + 1]
7                  A[j + 1] ← temp
```

В най-лошия случай сложността $T(n)$ на функцията `Sort` е следната:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i-1} 1 = \sum_{i=1}^{n-1} (n-i-1) = (n-2)+(n-3)+\cdots+0 = \frac{(n-2)(n-1)}{2} \asymp n^2.$$

По принцип $T(n)$ трябва да е сума от 4, а не от 1, но такъв константен брой операции, дори и приложени неконстантен брой пъти, не влияят на асимптотичното поведение.

Нека сега разгледаме следният алгоритъм за степенуване:

```
1  Exp(x, y ∈ ℕ):
2      res ← 1
3
4      while y > 0:
5          if y ≡ 1 (mod 2):
6              res ← res · x
7
8          y ← y/2
9          x ← x^2
10
11     return res
```

Той се възползва от простата идея, че за да сметнем да кажем 3^8, можем вместо 8 пъти да умножаваме числото 3, да представим 3^8 като $3^4 \cdot 3^4$. Тогава 3^4 можем да сметнем веднъж, и да го умножим със себе си. Пак можем да представим 3^4 като $3^2 \cdot 3^2$ и да пресметнем 3^2 само веднъж и да го умножим със себе си. Така при по-голяма стойност на y си спестяваме много работа.

С уговорката, че умножението е атомарна операция, сложността по време $T(n)$ (n е стойността на y) на функцията `Exp` е следната:

$$T(n) = \sum_{\substack{i=n \\ i \leftarrow \frac{i}{2}}}^{1} 1 = \underbrace{1 + \cdots + 1}_{\substack{\text{колкото пъти} \\ \text{можем да} \\ \text{делим целочислено} \\ n \text{ на 2 преди} \\ \text{да получим 0}}} = \underbrace{1 + \cdots + 1}_{\text{около } \log(n) \text{ пъти}} \asymp \log(n).$$

За да можем по-формално да изследваме този вид поведение, ще трябва да си поиграем малко с рекурентни уравнения.

2.4 Защо са ни рекурентни уравнения?

Те се появяват по естествен път, когато искаме да анализираме сложността на рекурсивни алгоритми.
Нека вземем за пример алгоритъма за двоично търсене:

```
1  BinarySearch(A[1...n] ∈ array(ℤ); l,r ∈ {1,...n}; v ∈ ℤ):
2      if l > r:
3          return −1
4
5      m ← ⌊(l+r)/2⌋
6
7      if A[m] = v: return m
8      if A[m] < v: return BinarySearch(A[1...n], m+1, r, v)
9      if A[m] > v: return BinarySearch(A[1...n], l, m−1, v)
```

При подаден сортиран целочислен масив $A[1 \dots n]$, негови индекси l, r и цяло число v, функцията $\texttt{BinarySearch}(A[1 \dots n], 1, n, v)$ ще върне индекс на $A[1 \dots n]$, в който се намира v, ако има такъв, иначе ще върне -1. Нека помислим каква е сложността на алгоритъма. Управляващите параметри на рекурсията са l и r. Всеки път разликата между двете намалява двойно (като накрая когато $l = r$ тя ще стане отрицателна).
Това означава, че в най-лошия случай сложността на алгоритъма може да се опише със следното рекурентно уравнение:

$T(0) = 2$ // заради ред 3 и 4

$T(n+1) = T(\lfloor \frac{n+1}{2} \rfloor) + 5$ // заради проверките и рекурсивното извикване

В този случай лесно се вижда асимптотиката на $T(n)$:

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + 5 \\ &= T(\lfloor \frac{n}{4} \rfloor) + 5 + 5 \\ &= T(\lfloor \frac{n}{8} \rfloor) + 5 + 5 + 5 = \cdots = T(0) + \underbrace{5 + \cdots + 5}_{\text{около } \log(n) \text{ пъти}} \asymp \log(n) \end{aligned}$$

Така получаваме, че алгоритъмът има сложност $O(\log(n))$. Обаче в общият случай далеч не е толкова лесно да се намери асимптотичното поведение на дадено рекурентно уравнение. Целта ни ще бъде да развием по-богат апарат за асимптотичен анализ на рекурентните уравнения.

2.5 Начини за намиране на асимптотиката на рекурентни уравнения

Начините се разделят на два типа:

- със решаване на уравнението;
- без решаване на уравнението.

И двата начина са ценни. Първият начин ни дава формула във явен вид, което може да ни е от полза. Понякога обаче формулата във явен вид не е *"красива"*, или изобщо не може да се намери такава. Тогава идва на помощ вторият начин. Той директно ни дава някаква *"хубава"* формула, без да трябва да намираме в явен вид решение на рекурентното уравнение. Проблема е обаче, че асимптотиката понякога е малко лъжлива – алгоритъм със сложност $2^{2^{2^{1000}}}$ е асимптотично по-бавен от алгоритъм със сложност n, но практически вторият е по-бърз.
Ще разгледаме следните методи (повечето от които са разглеждани по дискретна математика):

- налучкване и доказване
- развиване (което преди малко показахме)
- методът с характеристичното уравнение
- мастър-теоремата

Нека разгледаме един пример с налучкване:

$$T(0) = 3$$
$$T(n+1) = (n+1)T(n) - n$$

Започваме да разписваме:

n	$T(n)$	$n!$
0	3	1
1	3	1
2	5	2
3	13	6
4	49	24
5	241	120
6	1441	720

Вече лесно можем да покажем с индукция, че $T(n) = 2(n!) + 1$:

- В базата имаме, че $T(0) = 3 = 2 \cdot 1 + 1 = 2 \cdot 0! + 1$.
- За индуктивната стъпка:

$$\begin{aligned} T(n+1) &= (n+1)T(n) - n \overset{\text{(ИП)}}{=} (n+1)(2(n!)+1) - n \\ &= (n+1)(2(n!)) + n + 1 - n = 2(n+1)! + 1 \end{aligned}$$

Накрая получаваме, че $T(n) \asymp n!$
Нека сега да видим как можем да използваме метода на характеристичното уравнение:

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i) \ // \text{ функцията е добре дефинирана и за 0.}$$

Рекурентното уравнение, зададено в този вид, не може да се реши с този метод. За това ще трябва да направим преобразувания:

$$\begin{aligned} T(0) &= 1 \\ T(n+1) &= 1 + \sum_{i=0}^{n} T(i) = 1 + T(n) + \sum_{i=0}^{n-1} T(i) \\ &= T(n) + \underbrace{\left(1 + \sum_{i=0}^{n-1} T(i)\right)}_{T(n)} = 2T(n) + 1 = \underbrace{2T(n)}_{\text{хомогенна част}} \end{aligned}$$

Имаме само хомогенна част, от която получаваме характеристичното уравнение $x - 2 = 0$ с единствен корен 2. Така:

$$T(n) = A \cdot 2^n \text{ за някоя константи } A.$$

Вече няма нужда и да се намира константата – ясно е че $T(n) \asymp 2^n$. Като използваме метода на характеристичното уравнение, не е нужно да намираме накрая константите за да разберем каква е асимптотиката. Достатъчно е да вземем събираемото, която расте най-много. В случая е ясно, че това е 2^n.

Нека сега разгледаме и последният начин:

Теорема 2.5.1 (Мастър-теорема)**.** *Нека $a \geq 1$, $b > 1$ и $f \in \mathcal{F}$. Нека $T(n) = aT(\frac{n}{b}) + f(n)$, където $\frac{n}{b}$ се интерпретира като $\lfloor \frac{n}{b} \rfloor$ или $\lceil \frac{n}{b} \rceil$. Тогава:*

1 сл. Ако $f(n) \preceq n^{\log_b(a)-\varepsilon}$ за някое $\varepsilon > 0$, то тогава $T(n) \asymp n^{log_b(a)}$.

2 сл. Ако $f(n) \asymp n^{log_b(a)}$, то тогава $T(n) \asymp n^{log_b(a)} \log(n)$.

3 сл. Ако са изпълнени следните условия:

1. *$f(n) \succeq n^{\log_b(a)+\varepsilon}$ за някое $\varepsilon > 0$; и*
2. *съществува $0 < c < 1$, за което от някъде нататък $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$,*

то тогава $T(n) \asymp f(n)$.

Нека разгледаме рекурентното уравнение:

$$T(n) = 2T(\frac{n}{2}) + 1.$$

Тук $a = b = 2$, и $f(n) = 1$. Също така $\log_b(a) = 1$, откъдето $f(n) = 1 \preceq n^{\log_b(a)-\varepsilon}$, за $\varepsilon \in (0, 1)$. Така по 1 сл. на мастър-теоремата получаваме, че $T(n) \asymp n$.

2.6 Задачи

Задача 2.1. Да се намери асимптотиката на средната сложност по време на алгоритъма за бързо сортиране т.е. на рекурентното уравнение:

$$T(n) = \frac{1}{n}\left(\sum_{i=1}^{n-1} T(i) + T(n-i)\right) + n - 1.$$

Задача 2.2. Да се определи сложността по време за функцията:

```
1  F_1(n ∈ ℕ):
2      s ← 0
3
4      for i ← 1 to n:
5          for j ← 1; j ≤ i; j ← 2j:
6              s ← s + ij
7
8      return s
```

Задача 2.3. Да се определи сложността по време за функцията:

```
1  F_2(n ∈ ℕ):
2      s ← 0
3
4      for i ← 1; i² ≤ n; i ← i + 3:
5          s ← s + F_1(n)
6
7      return s
```

Задача 2.4. Да се определи сложността по време за функцията:

```
1  F_3(n ∈ ℕ):
2      s ← 0
3
4      for i ← 1; i ≤ n²; inc(i):
5          for j ← 1; j ≤ 2i; inc(j):
6              if j ≡ 0 (mod 2):
7                  s ← s + F_1(n)
8              else:
9                  s ← s + F_2(n)
10
11     return s
```

Задача 2.5. Да се намери асимптотиката на следните рекурентни уравнения:

$T_1(n) = 29T_1(\frac{n}{3}) + 2\sum_{i=1}^{n} \frac{1}{i^2}$

$T_2(n) = 29T_2(\frac{n}{3}) + 12n + \sqrt{n}$

$T_3(n) = T_3(n-1) + \frac{n}{(n+1)(n-1)}$

$T_4(n) = 29T_4(\frac{n}{3}) + (\sum_{i=1}^{n} \frac{1}{i})^4$

$T_5(n) = 29T_5(\frac{n}{3}) + 2\sum_{i=1}^{n} i^2$

$T_6(n) = 29T_6(\frac{n}{3}) + n^{\sqrt{n}} + (\sqrt{n})^n$

$T_7(n) = T_7(\sqrt{n}) + n$

$T_8(n) = 29T_8(\frac{n}{3}) + \binom{2n}{2}$

$T_9(n) = 8T_9(n-1) - T_9(n-2) + 2n2^{2n} + 3n2^{3n}.$

Задача 2.6. Да се намери сложността по време на следния алгоритъм:

```
1  𝔄₁(n ∈ ℕ):
2      if n < 2:
3          return n
4
5      a ← 0
6
7      for i ← 1 to n:
8          a ← a + i
9
10     return a + 𝔄₁(n − 1) + 𝔄₁(n − 1) + 𝔄₁(n − 2)
```

Ще се промени ли нещо ако връщаме $a + 2\mathfrak{A}_1(n-1) + \mathfrak{A}_1(n-2)$?

Задача 2.7. Да се намери сложността по време на следния алгоритъм:

```
1  𝔄₂(n ∈ ℕ):
2      if n < 2:
3          return 2
4
5      t ← 0
6      t ← t + 𝔄₂(n/3)
7
8      for i ← 2; i < n; i ← 2i:
9          t ← t + 1
10
11     t ← t · 𝔄₂(n/3)
12
13     return t
```

Задача 2.8. Да се намери асимптотиката на следните рекурентни уравнения:

$$T_1(n) = 2\sqrt{2}T_1\left(\frac{n}{\sqrt{2}}\right) + n^3 \qquad T_2(n) = T_2(n-1) + \frac{1+n}{n^2}$$

$$T_3(n) = \sum_{i=0}^{n-1}\left(T_3(i) + 2^{\frac{n}{2}}\right) \qquad T_4(n) = 5T_4\left(\frac{n}{2}\right) + n^2\log(n).$$

Глава 3

Коректност на алгоритми

3.1 Какво имаме предвид под коректност?

За целите на този курс един алгоритъм ще наричаме **коректен**, ако завършва при всякакви входни данни и връща правилен резултат при всякакви входни данни

Забележка. Въпреки че ние ще имаме това разбиране в курса, на практика тези изисквания невинаги са изпълнени:

- разглеждат се алгоритми, които могат и да не завършват за някои входни данни – от теоретична гледна точка са интересни за хората, които се занимават с теорията на изчислимостта;
- разглеждат се алгоритми, които много често (но не винаги) връщат правилния резултат – обикновено това се прави с цел бързодействие.

3.2 Едно *"ново"* понятие – инвариант

Специално за итеративните алгоритми се въвежда ново понятие - **инвариант**. Това са специални твърдения, свързани с цикъла.
В най-общият случай (за алгоритми) се формулират по следния начин:

"При k-тото достигане на ред l (ако има няколко инструкции казваме преди/след коя се намираме) в алгоритъма $\mathfrak{A}$ е изпълнено *някакво твърдение, зависещо от k и променливите, използвани в $\mathfrak{A}$*".

Доказателството на такива твърдения протича с добре познатата индукция. Първо доказваме базата т.е. какво се случва при първото достигане на цикъла. Индуктивното предположение и индуктивната стъпка се обединяват в *"нова"* фаза, наречена **поддръжка**. Довършителните разсъждения, които по принцип се намират след доказването на твърдението чрез индукция, ще наричаме **терминация**. Накрая показваме, че винаги ще излезнем от цикъла (**финитност**). Обикновено това ще го смятаме за очевидно (най-вече за `for`-цикли).

Внимание. Това, за което се използват инвариантите, е да се докаже коректността на ЕДИН цикъл, не на цял алгоритъм. Когато в алгоритъма ни има няколко цикъла, на всеки от тях трябва да съответства по един инвариант.

3.3 Инвариантите в действие

Нека разгледаме следния алгоритъм за степенуване на 2:

```
1  Pow2(n ∈ ℕ):
2      r ← 1
3
4      for i ← 1 to n:
5          r ← 2r
6
7      return r
```

Инвариант 3.3.1. *При всяко достигане на проверката за край на цикъла (на ред 4) е изпълнено, че* $r = 2^{i-1}$.

Доказателство.
База. Наистина при първото достигане имаме, че $i = 1$ и от там $r = 1 = 2^{i-1}$.
Поддръжка. Нека при някое непоследно достигане твърдението е изпълнено. Тогава преди следващото достигане на проверката на r присвояваме $2r$, като знаем, че преди r е бил 2^{i-1}, и след това на i присвояваме $i + 1$. Така е ясно, че при новото достигане на проверката r ще стане $2 \cdot 2^{i_{old}-1} = 2^{i_{old}} = 2^{i-1}$.
Терминация. Ако е изпълнено условието за край на цикъла, то тогава $i = n+1$, откъдето ще върнем $r = 2^{(n+1)-1} = 2^n$.
Финитност. Величината $n - i$ започва с $n - 1$, и намалява с 1, докато не стигне -1, когато ще излезнем от цикъла. Следователно алгоритъмът винаги завършва.
□

3.4 С инвариантите трябва да се внимава

Един от често срещаните капани, в които попадат хората, е да не си формулират инвариантът добре. Много е важно инвариант да дава достатъчна информация за това което наистина се случва в алгоритъма. За целта ще разгледаме един пример:

```
1  SelectionSort(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n − 1:
3          m ← i
4
5          for j ← i + 1 to n:
6              if A[j] < A[m]:
7                  m ← j
8
9          swap(A[i], A[m])
```

На интуитивно ниво е ясно какво прави кода. Намира най-малкия елемент, и го слага на първо място. След това намира втория най-малък елемент, и го слага на второ място, и т.н.
Нещо, което някои биха се пробвали да направят за първия цикъл, е следното:

При всяко достигане на проверката за край на цикъла на ред 3 *подмасивът* $A[1 \dots i-1]$ *е сортиран.*

Проблемът с това твърдение, е че може много лесно да се измисли алгоритъм, за който това твърдение е изпълнено, и изобщо не сортира елементите в масива:

```
1  TrustMeItSorts(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n:
3          A[i] ← i
```

Очевидно този за този алгоритъм горната инвариант е изпълнена, но той е безсмислен. Получаваме сортиран масив, но за сметка на това губим цялата информация, която сме имали за него.
Нещо друго, което е важно да се направи, е първо да се формулира инвариант за вътрешния цикъл, и после за външния, като тънкият момент тук е, че ще ни трябват допускания за първият инвариант. Идеята е, че външния цикъл разчита на вътрешния да си свърши работата, и обратно вътрешния разчита (не винаги) на външния преди това да си е свършил работата.

Нека покажем как трябва да станат инвариантите, като доказателството оставяме за упражнение на читателя. Нека $A^*[1 \dots n]$ е първоначалната стойност на входния масив.

Инвариант 3.4.1 (вътрешен цикъл)**.** *При всяко достигане на проверката за край на цикъла на ред* 5 *имаме, че* m *е индексът на най-малкия елемент в масива* $A[i \dots j-1]$.

Инвариант 3.4.2 (външен цикъл)**.** *При всяко достигане на проверката за край на цикъла на ред* 2 *имаме, че масивът* $A[1 \dots i-1]$ *съдържа сортирани първите* $i-1$ *по големина елементи на* $A^*[1 \dots n]$*, като останалите са в* $A[i \dots n]$.

Обикновено в доказателството на коректност на алгоритми най-трудното е да се формулира инвариантът. Ако човек има добре формулирана инвариант, доказателството е на първо място възможно, а на второ – по-лесно.

3.5 Подход при задачи с вече даден алгоритъм

Нека разгледаме следния алгоритъм:

```
1  Foo(a ∈ ℕ):
2      x ← 6
3      y ← 1
4      z ← 0
5
6      for i ← 0 to a − 1:
7          z ← z + y
8          y ← y + x
9          x ← x + 6
10
11     return z
```

Питаме се какво връща той?
Обикновено в такива задачи трябва да се изпробва алгоритъма върху няколко стойности. Можем да забележим, че:

- `Foo`(0) връща 0;
- `Foo`(1) връща 1;
- `Foo`(2) връща 8 и така нататък.

Вече можем да видим какво прави алгоритъмът – `Foo`(a) връща a^3. Нека сега докажем това:

Инвариант 3.5.1. *При всяко достигане на проверката за край на цикъла на ред* 5 *е изпълнено, че:*

- $x = 6(1 + i)$;
- $y = 3i^2 + 3i + 1$;
- $z = i^3$.

Доказателство.
База. При първото достигане имаме, че:

- $i = 0$;
- $x = 6 = 6 \cdot 1 = 6(1 + 0) = 6(1 + i)$;
- $y = 1 = 3 \cdot 0^2 + 3 \cdot 0 + 1 = 3i^2 + 3i + 1$;
- $z = 0 = 0^3 = i^3$.

Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава при влизане в тялото на цикъла:

- z ще стане $z + y \overset{\text{(ИП)}}{=} i^3 + 3i^2 + 3i + 1 = (\underbrace{i + 1}_{\text{ново } i})^3$;
- y ще стане $y + x \overset{\text{(ИП)}}{=} 3i^2 + 3i + 1 + 6 + 6i = 3(\underbrace{i + 1}_{\text{ново } i})^2 + 3(\underbrace{i + 1}_{\text{ново } i}) + 1$;
- x ще стане $x + 6 \overset{\text{(ИП)}}{=} 6(1 + i) + 6 = 6(1 + \underbrace{i + 1}_{\text{ново } i})$.

Терминация. В последното достигане на проверката за край на цикъла имаме, че $i = a$, и тогава на ред 12 алгоритъмът ще върне $z = a^3$.
Финитност. (от тук нататък повечето няма да ги пишем) Величината $a - i$ започва с a, и намалява с 1, докато не стигне 0, когато ще излезнем от цикъла. Следователно алгоритъмът винаги завършва. □

Нека разгледаме още един такъв пример:

```
1  bar(n ∈ ℤ): return √(n²)
2
3  foo(x, y ∈ Z): return (x+y+bar(x−y))/2
4
5  quux(A[1...n] ∈ array(ℤ)):
6      a ← A[1]
7
8      for i ← 2 to n:
9          a ← foo(a, A[i])
10
11     return a
```

Искаме да видим какво връща $\mathfrak{quux}(A[1 \dots n])$. Тук най-трудното, което трябва да се направи, е да се определи какво връщат $\mathfrak{bar}$ и $\mathfrak{foo}$:

- $\mathfrak{bar}(n) = \sqrt{n^2} = |n|$;
- $\mathfrak{foo}(x, y) = \frac{x+y+|x-y|}{2} = \max\{x, y\}$:
  - ако $x \geq y$, то $\frac{x+y+|x-y|}{2} = \frac{x+y+x-y}{2} = \frac{2x}{2} = x = \max\{x, y\}$,
  - ако $x < y$, то $\frac{x+y+|x-y|}{2} = \frac{x+y+y-x}{2} = \frac{2y}{2} = y = \max\{x, y\}$.

Като знаем какво правят $\mathfrak{bar}$ и $\mathfrak{foo}$, можем да забележим, че $\mathfrak{quux}(A[1 \dots n])$ дава най-големия елемент на $A[1 \dots n]$:

Инвариант 3.5.2. *При всяко достигане на проверката за край на цикъла на ред* 15 *е изпълнено, че* $a = \max A[1 \dots i-1]$.

Доказателство.
База. Наистина при първото достигане $a = A[1] = \max A[1 \dots i-1]$.
Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава като влезем в тялото на цикъла, променливата a става:

$$\begin{aligned}\mathfrak{foo}(a, A[i]) \overset{(\text{ИП})}{=} \mathfrak{foo}(\max A[1 \dots i-1], A[i]) &= \max(\max A[1 \dots i-1], A[i]) \\ &= \max A[1 \dots i] = \max A[1 \dots \underbrace{i+1}_{\text{ново } i} - 1]\end{aligned}$$

Терминация. При последното достигане на проверката за край на цикъла на ред 15 променливата i ще бъде $n+1$, откъдето функцията ще върне точно $a = \max A[1 \dots (n+1)-1] = \max A[1 \dots n]$, с което сме готови. □

3.6 Примери за рекурсивни алгоритми

Нека разгледаме следният алгоритъм:

```
1  𝔐(A[1...n] ∈ array(ℤ)):
2      if n = 0:
3          return −∞
4      else:
5          return max(A[n], 𝔐(A[1...n − 1]))
```

Очевидно при параметър целочислен масив $A[1 \dots n]$, функцията $\mathfrak{M}(A[1 \dots n])$ връща $\max A[1 \dots n]$. Ще докажем това с индукция по n:

- В базата имаме, че $\mathfrak{M}(A[1 \dots n]) = -\infty = \max[] = \max A[1 \dots 0]$ където със [] означаваме празният масив.
- За индуктивната стъпка имаме, че:

$$\begin{aligned}\mathfrak{M}(A[1 \dots n+1]) &= \max(A[n+1], \mathfrak{M}(A[1 \dots n])) \overset{(\text{ИП})}{=} \max(A[n+1], \max A[1 \dots n]) \\ &= \max A[1 \dots n+1].\end{aligned}$$

Тук управляващият параметър на рекурсията n винаги намалява с 1, докато не стигне 0, където ще приключи алгоритъмът. В по нататъчните разсъждения ще смятаме завършването на алгоритъма за очевидно.
Сложността на алгоритъма се описва със рекурентното уравнение:

$$T(n) = T(n-1) + 1 \ // \text{ базата няма да я пишем.}$$

Директно се вижда, че $T(n) = \sum_{i=0}^{n} 1 = n + 1 \asymp n$.
Да видим един малко по-сложен пример – за бързо степенуване:

```
1  𝒫(x, y ∈ ℕ):
2      if y = 0:
3          return 1
4
5      s ← 𝒫(x, y/2)
6
7      if y ≡ 1 (mod 2):
8          return s²x
9      else:
10         return s²
```

С пълна индукция относно y ще покажем, че $\mathcal{P}(x, y) = x^y$:

- $\mathcal{P}(x, 0) = 1 = x^0 = 1$ // тук се уговаряме, че $0^0 = 1$.
- $\mathcal{P}(x, 2y+1) = x \cdot \mathcal{P}(x, y) \cdot \mathcal{P}(x, y) \overset{\text{(ИП)}}{=} x \cdot x^y \cdot x^y = x^{2y+1}$.
- $\mathcal{P}(x, 2y+2) = \mathcal{P}(x, y+1) \cdot \mathcal{P}(x, y+1) \overset{\text{(ИП)}}{=} x^{y+1} \cdot x^{y+1} = x^{2y+2}$.

Сложността на този алгоритъм може да се опише със следното рекурентно уравнение:

$$T(n) = T(\frac{n}{2}) + 1.$$

От мастър-теоремата следва, че:

$$T(n) \asymp \log(n).$$

3.7 Трик за бързо пресмятане на членове на някои рекурентни редици

Нека вземем за пример редицата на Фибоначи:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n+2) &= F(n+1) + F(n) \end{aligned}$$

Човек може да забележи, че:

$$\underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}}_{\mathfrak{F}^*} \cdot \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} F(n+2) \\ F(n+1) \end{pmatrix}.$$

След това с индукция лесно се показва, че:

$$(\mathfrak{F}^*)^n \cdot \begin{pmatrix} F(1) \\ F(0) \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix}.$$

За да сметнем $F(n)$, можем да направим бързо степенуване на матрицата, аналогична на тази с числата:

```
1  FibMatrixExp(R ∈ (ℕ)_{2×2}; n ∈ ℕ):
2      if n = 0:
3          R ← E_{2×2}
4
5      FibMatrixExp(R, n/2)
6      R ← R^2
7
8      if n ≡ 1 (mod 2):
9          R ← R · 𝔉*
10
11 F(n ∈ ℕ):
12     if n ≤ 1:
13         return n
14
15     init(R ∈ (ℕ)_{2×2})
16     FibMatrixExp(R, n − 1);
17
18     return R[1][1];
```

Коректността и сложността на алгоритъма оставяме на читателя (напълно аналогично е на предния алгоритъм).
В общия случай ще имаме рекурентно уравнение от вида:

$T(n+k+1) = a_k T(n+k) + \cdots + a_1 T(n+1) + a_0 T(n)$, където $a_0, \ldots, a_k, k$ са константи.

Тогава отново с индукция човек лесно може да покаже, че:

$$\begin{pmatrix} a_k & a_{k-1} & a_{k-2} & \ldots & a_2 & a_1 & a_0 \\ 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} T(k) \\ T(k-1) \\ T(k-2) \\ \vdots \\ T(0) \end{pmatrix} = \begin{pmatrix} T(n+k) \\ T(n+k-1) \\ T(n+k-2) \\ \vdots \\ T(n) \end{pmatrix}.$$

3.8 Още примери

Искаме да напишем алгоритъм, който при подадена двойка от естествени числа $\langle a, b\rangle$, където $b \leq a$, да върне $\gcd(a, b)$ т.е. това число $d \in \mathbb{N}$, за което:

- $d \mid a$ и $d \mid b$;
- ако $d_1 \mid a$ и $d_1 \mid b$, то $d_1 \mid d$.

За целта ще покажем следното:

Твърдение 3.8.1. *За всяко $a, b, q, r \in \mathbb{N}$, където $0 < b \leq a$, $a = bq + r$ и $r \in \{0, \ldots, b-1\}$, е изпълнено, че:*

$$\gcd(a, b) = \gcd(b, r)$$

Доказателство. Първо имаме, че $\gcd(a, b) \mid a$ и $\gcd(a, b) \mid b$, откъдето понеже $a = bq + r$, имаме $\gcd(a, b) \mid r$. Тогава $\gcd(a, b) \mid \gcd(b, r)$. От друга страна $\gcd(b, r) \mid b$ и $\gcd(b, r) \mid r$, откъдето $\gcd(b, r) \mid bq + r = a$. Така $\gcd(b, r) \mid \gcd(a, b)$. Накрая можем да заключим $\gcd(a, b) = \gcd(b, r)$ от това, че $\gcd(a, b) \mid \gcd(b, r)$ и $\gcd(b, r) \mid \gcd(a, b)$. □

На базата на това наблюдение се получава следният алгоритъм (кръстен на Евклид):

```
1  Euclid(a, b ∈ ℕ where a ≥ b):
2      if b = 0:
3          return a
4
5      return Euclid(b, mod(a, b))
```

Ще покажем с индукция относно b, че за всяко $a \geq b$, е изпълнено, че:

$$\texttt{Euclid}(a, b) = \gcd(a, b)$$

- В базовия случай имаме $\texttt{Euclid}(a, 0) = a = \gcd(a, 0)$.
- Нека $b > 0$ и $a = bq + r$ за някои $r \in \{0, \ldots, b-1\}$ и q и нека твърдението е изпълнено за всяко $b' < b$. Тогава:

$$\texttt{Euclid}(a, b) = \texttt{Euclid}(b, r) \overset{\text{(ИП)}}{=} \gcd(b, r) \overset{\text{Твърдение 3.8.1}}{=} \gcd(a, b).$$

Алгоритъмът терминира, понеже управляващият параметър `b` винаги намаля, докато не стане `0`. На пръсти ще покажем, че алгоритъмът има сложност по време и памет $O(\log(a))$. Нека видим, че ако $a > b$, то $\mathrm{mod}(a, b) < \frac{a}{2}$:

1 сл. ако $b \leq \frac{a}{2}$, то $\mathrm{mod}(a, b) < \frac{a}{2}$ и сме готови;

2 сл. ако пък $b > \frac{a}{2}$, то тогава $a - b < \frac{a}{2}$, откъдето $\mathrm{mod}(a, b) = \mathrm{mod}(a - b, b) = a - b < \frac{a}{2}$.

Тогава през всеки две стъпки на алгоритъма от вход $\langle a,\ b \rangle$, отиваме до вход $\langle\ \mathrm{mod}\ (a,\ b),\ \mathrm{mod}\ (b,\ \mathrm{mod}\ (a,\ b))\rangle$ и левият аргумент става поне два пъти по-малък. Тъй като левият аргумент е горна граница за десния, то той също ще намаля рязко. Дълбочината на дървото на рекурсията зависи само от броя на рекурсивните извиквания. Понеже те са логаритмично много и всяко едно от тях заема константна памет, сложността по памет е също $O(\log(a))$.

Вторият пример е задача за числа, скрита в задача за масиви:

Имаме един масив $A[1 \ldots n]$, който съдържа всички числа от 1 до n, с изключение на едно от тях, което е заместено с друго число от 1 до n. Искаме да напишем колкото се може по-бърз алгоритъм, който при вход такъв масив $A[1 \ldots n]$ връща наредената двойка $\langle d,\ m \rangle$ от дупликата и липсващото число. Оказва се, че тази задача може да се реши за линейно време и константна памет. За целта трябва да се забележат следните факти:

- $d - m = \sum_{i=1}^{n} A[i] - \sum_{i=1}^{n} i = \sum_{i=1}^{n} A[i] - \frac{n(n+1)}{2} =: B$;
- $(d + m)(d - m) = d^2 - m^2 = \sum_{i=1}^{n} A[i]^2 - \sum_{i=1}^{n} i^2 = \sum_{i=1}^{n} A[i]^2 - \frac{n(n+1)(2n+1)}{6} =: C$.

От тях получаваме следната система от уравнения:

$$\left| \begin{array}{l} d - m = B \\ d + m = \frac{C}{B} \end{array} \right.$$

Най-сложното, което трябва да направим, е да пресметнем B и C:

```
1  FindMissingAndDuplicate(A[1...n]):
2      (B, C) ← (A[1] + ··· + A[n] − 1 − ··· − n, A[1]^2 + ... A[n]^2 − 1^2 − ··· − n^2)
3
4      d ← (B + C/B)/2
5      m = d − B
6
7      return (d, m)
```

Нека за пълнота покажем следното:

Твърдение 3.8.2. *За всяко $n \in \mathbb{N}$ е изпълнено, че:*

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \text{ и } \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Доказателство. Правим индукция по n:

- В базата имаме, че $\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$ и $\sum_{i=0}^{0} i^2 = \frac{0(0+1)(2\cdot 0+1)}{6}$.
- В стъпката имаме, че:

  $\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1) \overset{\text{(ИП)}}{=} \frac{n(n+1)}{2} + (n+1) = \frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$, и

  $$\begin{aligned}\sum_{i=0}^{n+1} i^2 &= \sum_{i=0}^{n} i^2 + (n+1)^2 \overset{\text{(ИП)}}{=} \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \\ &= \frac{n(n+1)(2n+1)}{6} + \frac{6(n+1)^2}{6} \\ &= \frac{(n+1)(2n^2+n+6n+6)}{6} = \frac{(n+1)(n+2)(2(n+1)+1)}{6}.\end{aligned}$$

□

3.9 Задачи

Задача 3.1. Даден е следният алгоритъм:

```
1  𝔄(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n − 1:
3          for j ← i + 1 to n:
4              if A[i] = A[j]:
5                  return 𝕋
6
7      return 𝔽
```

1. Какво връща той? Отговорът да се обоснове.
2. Каква е неговата сложност по време и памет?

Задача 3.2. Даден е следният алгоритъм:

```
1  𝔉(n ∈ ℕ):
2      if n < 2:
3          return n
4
5      a ← 0
6      b ← 1
7
8      for i ← 1 to n − 1:
9          t ← a
10         a ← b
11         b ← t + b
12
13     return b
```

Да се докаже, че $\mathfrak{F}(n)$ връща n-тото число на Фибоначи.

Задача 3.3. Даден е следният: алгоритъм:

```
1  Mult(A, B, C ∈ (ℤ)_{n×n})
2      for i ← 1 to n:
3          for j ← 1 to n:
4              s ← 0
5
6              for k ← 1 to n:
7                  s ← s + A[i][k] · B[k][j]
8
9              C[i][j] ← s
```

Да се докаже, че при вход $n \times n$ целочислени матрици A, B и C функцията $\texttt{Mult}(A, B, C)$ записва в C произведението на A и B. Да се намери сложността му по време и памет.

Задача 3.4. Да се напише колкото се може по-бърз алгоритъм, който при подадено крайно множество от точки $P \subseteq \mathbb{Z} \times \mathbb{Z}$, намира

$$\max\{|x_1 - x_2| + |y_1 - y_2| \mid (x_1, y_1), (x_2, y_2) \in P\}.$$

След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 3.5. Даден е следният алгоритъм:

```
1  NumSlopes(A[1...n] ∈ array(ℤ)):
2      s ← 1
3
4      for i ← 2 to n:
5          if A[i − 1] > A[i]:
6              s ← s + 1
7
8      return s
```

Какво връща `NumSlopes`$(A[1 \dots n])$? Отговорът да се обоснове.

Задача 3.6. Даден е следният алгоритъм:

```
1  Kadane(A[1...n] ∈ array(ℤ)):
2      max_so_far ← A[1]
3      max_ending_here ← A[1]
4
5      for i ← 2 to n:
6          max_ending_here = max(max_ending_here + A[i], A[i])
7          max_so_far = max(max_ending_here, max_so_far)
8
9      return max_so_far
```

Какво връща `Kadane`$(A[1 \dots n])$? Отговорът да се обоснове.

Задача 3.7. Да се напише алгоритъм `Calculate`$(F[0 \dots k], S[0 \dots k], n)$, който приема два целочислени масива $F[0 \dots k]$ и $S[0 \dots k]$, естествено число n и връща числото $T(n)$, където:

$$
\begin{aligned}
T(0) &= F[0] \\
T(1) &= F[1] \\
&\vdots \\
T(k) &= F[k] \\
T(n+k+1) &= S[k] \cdot T(n+k) + \dots + S[1] \cdot T(n+1) + S[0] \cdot T(n).
\end{aligned}
$$

След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 3.8. Даден е следният алгоритъм:

```
1  FindMajority(A[1...n] ∈ array(ℤ)):
2      m ← A[1]
3      c ← 1
4
5      for i ← 2 to n:
6          if c = 0:
7              m ← A[i]
8              c ← 1
9          else if A[i] = m:
10             c ← c + 1
11         else:
12             c ← c − 1
13
14     return m
```

Да се докаже, че при подаден целочислен масив $A[1 \dots n]$, в който има елемент с повече от $\lfloor \frac{n}{2} \rfloor$ срещания, функцията `FindMajority`$(A[1 \dots n])$ ще върне точно този елемент.

Задача 3.9. Да се напише алгоритъм `IsDerivationTree`$(G = \langle \Sigma, V, S, R \rangle, T)$, който приема безконтекстна граматика G, дърво T и проверява дали T е дърво на извод за G. След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 3.10. Масив на Монж ще наричаме всеки двумерен масив $A[1 \dots n, 1 \dots m]$ от естествени числа, за който:

$$A[p, q] + A[s, t] \leq A[p, t] + A[s, q] \text{ за всички } 1 \leq p, s \leq n \text{ и } 1 \leq q, t \leq n.$$

Да се напише алгоритъм със линейна сложност (т.е. $\Theta(n \cdot m)$), който приема двумерен масив $A[1 \dots n, 1 \dots m]$ и проверява дали е масив на Монж. След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 3.11. Да се напише алгоритъм, който приема естествено число n и връща булев масив $P[2 \dots n]$, за който е изпълнено, че за всяко $2 \leq i \leq n$:

$$P[i] \text{ е истина} \iff i \text{ е просто.}$$

След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 3.12. Даден е следният алгоритъм:

```
1  SS(A[1...n]; l, h ∈ {1,...,n}):
2      if l ≥ h:
3          return
4
5      if A[l] > A[h]:
6          swap(A[l], A[h])
7
8      t ← (h−l+1)/3
9
10     if t ≥ 1:
11         SS(A[1...n], l, h − t)
12         SS(A[1...n], l + t, h)
13         SS(A[1...n], l, h − t)
```

Какво прави $\texttt{SS}(A[1 \dots n], 1, n)$ и каква е сложността на този алгоритъм по време и памет? Обосновете отговорите си.

Задача 3.13. Да се напише колкото се може по-бърз алгоритъм, който приема целочислен масив $A[1 \dots n]$ и връща масив $B[1 \dots n]$, такъв че за всяко $1 \leq i \leq n$:

$$B[i] = \prod_{\substack{k=1 \\ k \neq i}}^{n} A[i].$$

След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 3.14. Да се напише колкото се може по-бърз алгоритъм, който приема целочислен масив $A[1 \dots n]$ и връща двойка $\langle i, j \rangle$ от различни индекси, за която:

$$A[i] \cdot A[j] = \max_{1 \leq i' < j', \leq n} A[i'] \cdot A[j'].$$

След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 3.15. Като вход е даден масив от естествени числа $H[1 \dots n]$, който представя линии между точките $(i, 0)$ и $(i, H[i])$. Да се напише алгоритъм, който намира индексите на двете линии, които образуват контейнер, който би събрал максимално много вода.

Задача 3.16. Даден е следният алгоритъм:

```
1  Ц(A[1...n] ∈ array(ℤ)):
2      (x, y) ← (𝕋, 𝕋)
3
4      for i ← 1 to n − 1:
5          if x & A[i] > A[i + 1]:
6              x ← 𝔽
7          else if ¬x & A[i] < A[i + 1]:
8              y ← 𝔽
9
10     return y
```

Какво връща $Ц(A[1 \dots n])$? Обосновете отговора си.

Задача 3.17. Дадени са n на брой бензиностанции, които са подредени в кръгов маршрут. Масивът $G[1 \dots n]$ съдържа литрите бензин в бензиностанциите, а масивът $C[1 \dots n]$ съдържа цената (в литри бензин) за придвижване от една бензиностанция до следващата (след бензиностанция n е бензиностанция 1). Приемаме, че имаме неограничен капацитет на резервоара. Да се състави алгоритъм, който връща индекс на бензиностанция, от която, започвайки с празен резервоар, може да се мине през всички бензиностанции. Ако не съществува такъв индекс, да се върне -1. За простота може да си мислите, че ако има такъв индекс, ще бъде единствен.

Глава 4

Сортиране и неговите приложения

4.1 Каква задача решаваме?

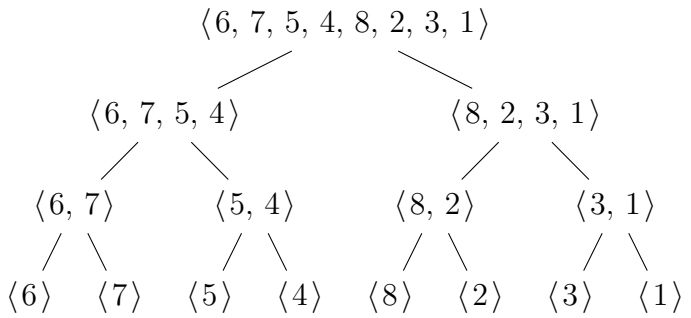
Ще формулираме задачата във по-общ вариант. Имаме някаква преднаредба (рефлексивна и транзитивна релация) $\leq_A$ върху множество A. Даден ни е масив $B[1 \ldots n]$ със елементи от A и търсим пермутация $B'[1 \ldots n]$ на $B[1 \ldots n]$, за която:

$$B'[1] \leq_A B'[2] \leq_A \cdots \leq_A B'[n].$$

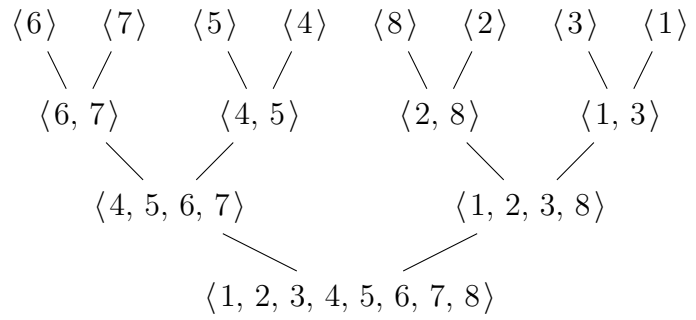
На нас обикновено ще ни интересува частния случай $(A, \leq_A) = (\mathbb{Z}, \leq)$. Ще разгледаме малко по-подробно два сортиращи алгоритъма – за сортиране чрез сливане и за бързо сортиране.

4.2 Сортиране чрез сливане

Нека си представим, че искаме да сортираме редицата $\langle 6, 7, 5, 4, 8, 2, 3, 1 \rangle$. Можем да направим следното разделение на подзадачи за сортиране:



Сега това, което трябва да направим, е да започнем да сливаме масивите:



Ако имахме функция `Merge`, която приемаше два сортирани масива $A[1 \dots n], B[1 \dots m]$ и връщаше масив $C[1 \dots n+m]$, който съдържа елементите на $A[1 \dots n]$ и $B[1 \dots m]$ в сортиран ред, то тогава щяхме да получим следния алгоритъм за сортиране:

```
1  MergeSort(A[1...n] ∈ array(ℤ)):
2      if n = 1:
3          return copy(A[1...n])
4
5      MergeSort(A[1...⌊n/2⌋])
6      MergeSort(A[⌊n/2⌋ + 1...n])
7      return Merge(A[1...⌊n/2⌋], A[⌊n/2⌋ + 1...n])
```

Ще докажем това с индукция по n:

- В базовия случай $n = 1$. Всеки едноелементен масив е сортиран и ние коректно ще върнем копие на $A[1 \dots n]$.
- Нека сега $n > 1$. По (ИП) двете извиквания на `MergeSort` ще сортират $A[1 \dots \lfloor \frac{n}{2} \rfloor]$ и $A[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$. След това по допускане извикването на `Merge` коректно ще ги слее в нов сортиран масив и след това алгоритъмът ще върне този масив.

Ако `Merge` завършва за време $T_M(n)$, то сложността на `MergeSort` може да се опише с рекурентното уравнение:

$$T(n) = 2T(\frac{n}{2}) + T_M(n).$$

Нещо повече, ако покажем алгоритъм `Merge` със линейна сложност по време, то тогава:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) \asymp n \log(n).$$

Ето алгоритъм със сложност $\Theta(n+m)$, който слива два сортирани масива:

```
1  Merge(A[1...n], B[1...m] ∈ array(ℤ)):
2      init C[1...n+m] ∈ array(ℤ)
3      (i, l_A, l_B) ← (1, 1, 1)
4
5      while (l_A ≤ n & l_B ≤ m):
6          if A[l_A] ≤ B[l_B]:
7              C[i] ← A[l_A]
8              l_A ← l_A + 1
9          else:
10             C[i] ← A[l_B]
11             l_B ← l_B + 1
12         i ← i + 1
13
14     while l_A ≤ n:
15         C[i] ← A[l_A]
16         l_A ← l_A + 1
17         i ← i + 1
18
19     while l_B ≤ m:
20         C[i] ← A[l_B]
21         l_B ← l_B + 1
22         i ← i + 1
23
24     return C[1...n+m]
```

Нека сега формулираме инвариантът (той ще бъде един за всички цикли), чието доказателство оставяме на читателя:

Инвариант 4.2.1. *При всяко достигане на условието за край на цикъла на редове* $7, 16$ *и* 21 *е изпълнено, че:*

- $C[1 \dots i-1]$ *съдържа елементите на* $A[1 \dots l_A - 1]$ *и* $B[1 \dots l_B - 1]$*, и то в сортиран ред;*
- l_A *е индексът на най-малкия елемент на* $A[1 \dots n]$*, който още не е копиран в* $C[1 \dots n+m]$*;*
- l_B *е индексът на най-малкия елемент на* $B[1 \dots m]$*, който още не е копиран в* $C[1 \dots n+m]$*.*

4.3 Бързо сортиране

Сега ще покажем още една идея за сортиране. Нека отново ни е дадена редицата $\langle 6, 7, 5, 4, 8, 2, 3, 1\rangle$. Ако можехме да пренаредим масива така, че всички елементи, които са по-малки от 5 да са вляво от 5 и останалите да бъдат вдясно от 5, то тогава щяхме да трябва да сортираме двете по-малки редици. Да кажем, след пренареждането получаваме редицата $\langle 4, 2, 1, 3, 5, 7, 6, 8\rangle$. Тогава ще трябва само да сортираме редиците $\langle 4, 2, 1, 3\rangle$ и $\langle 7, 6, 8\rangle$. По формално, нека си представим, че имаме алгоритъм `Partition`, която приема целочислен масив $A[1 \dots n]$ и го пренарежда така, че да има индекс pp, за което:

- $A[i] < A[pp]$ за всяко $1 \leq i < pp$;
- $A[i] \geq A[pp]$ за всяко $pp \leq i \leq n$,

и след това връща този индекс pp.
Тогава можем да получим сортиращ алгоритъм по следния начин:

```
1  Quicksort(A[1...n] ∈ array(ℤ)):
2      if n ≤ 1:
3          return
4
5      pp ← Partition(A[1...n])
6      Quicksort(A[1...pp − 1])
7      Quicksort(A[pp + 1...n])
```

Да докажем, че `Quicksort`$(A[1 \dots n])$ сортира $A[1 \dots n]$ с индукция по n:

- В базовия случай $n \leq 1$, откъдето масивът е сортиран.
- Нека сега $n > 1$. По допускане след извикването на `Partition`:
  - $A[i] < A[pp]$ за всяко $1 \leq i < pp$;
  - $A[i] \geq A[pp]$ за всяко $pp \leq i \leq n$,

  Тогава по (ИП) алгоритъмът коректно ще сортира масивите $A[1 \dots pp - 1]$ и $A[pp + 1 \dots n]$.

Ще покажем, че `Partition` може да се имплементира със сложност $\Theta(n)$. Тук обаче сложността на сортиращия алгоритъм зависи изцяло от това какво е pp. В най-лошия случай $pp = 1$ или $pp = n$ и тогава ще имаме рекурентното уравнение:

$$T(n) = T(n-1) + T(1) + \Theta(n) \asymp n^2.$$

В средния случай се оказва, че сложността е $n\log(n)$. На читателя оставяме да покаже, че:

$$T(n) = \frac{1}{n}\left(\sum_{k=1}^{n-1} T(k) + T(n-k)\right) + \Theta(n) \asymp n\log(n).$$

Остава само да направим имплементация на `Partition` със сложност $\Theta(n)$:

```
1  Partition(A[1...n] ∈ array(ℤ)):
2      pivot ← A[n]
3      pp ← 1
4
5      for i ← 1 to n − 1:
6          if A[i] < pivot:
7              swap(A[i], A[pp])
8              pp ← pp + 1
9
10     swap(A[pp], A[n])
11     return pp
```

На читателя оставяме доказателството на следния:

Инвариант 4.3.1. *При всяко достигане на проверката за край на цикъла на ред* 5 *е изпълнено, че:*

- $A[j] < pivot$ *за всяко* $1 \leq j < pp$*;*
- $A[j] \geq pivot$ *за всяко* $pp \leq j < i$*.*

Функцията `Partition` е много удобна. Чрез нея можем да намираме k-ти по големина елемент на масив с уникални елементи по следния начин:

```
1  Select(A[1...n] ∈ array(ℤ); k ∈ {1,...,n}):
2      pp ← Partition(A[1...n])
3
4      if k = pp:
5          return A[k]
6      if k < pp:
7          return Select(A[1...pp − 1], k)
8      if k > pp:
9          return Select(A[pp + 1...n], k − pp)
```

Ще докажем, че `Select`$(A[1\ldots n], k)$ връща k-тия по големина елемент на $A[1\ldots n]$ с индукция относно n:

- В базовия случай $n = 1$, откъдето след извикването на `Partition` ще имаме $pp = k = 1$ и алгоритъмът коректно ще върне $A[k]$.
- Нека сега $n > 1$. След извикването на `Partition`:
  - $A[i] < A[pp]$ за всяко $1 \leq i < pp$;
  - $A[i] \geq A[pp]$ за всяко $pp \leq i \leq n$.

  Нека сега разгледаме трите случая:

1 сл. Ако $k = pp$, то понеже $A[pp]$ е pp-тия по големина елемент, алгоритъмът ще върне правилният отговор.

2 сл. Ако $k < pp$, то е достатъчно да намерим k-тия по големина елемент на масива $A[1\ldots pp-1]$, тъй като там са елементите, по-малки от $A[pp]$. По (ИП) ще получим коректен отговор.

3 сл. Ако $k > pp$, то е достатъчно да намерим $(k-pp)$-тия (понеже махаме първите pp елемента на $A[1\ldots n]$) по големина елемент на масива $A[pp+1\ldots n]$, тъй като там са елементите, по-големи от $A[pp]$. По (ИП) ще получим коректен отговор.

Тук отново сложността зависи от стойността на pp, която получаваме. За съжаление, отново в най-лошия случай имаме:

$$T(n) = T(n-1) + n \asymp n^2.$$

В средния случай получаваме:

$$T(n) = \frac{1}{n}\sum_{k=1}^{n-1} T(k) + n.$$

Сега ще покажем, че в средния случай:

$$T(n) \asymp n.$$

Първо, можем да умножим по n и от двете страни и получаваме:

$$nT(n) = \sum_{k=1}^{n-1} T(k) + n^2.$$

Тогава ако заместим n с $n-1$ и извадим, ще получим:

$$nT(n) - (n-1)T(n-1) = T(n-1) + n^2 - (n-1)^2 = T(n-1) + 2n - 1.$$

Но това е еквивалентно на:

$$nT(n) = nT(n-1) + 2n - 1.$$

Сега разделяме на n и получаваме:

$$T(n) = T(n-1)+2-\frac{1}{n} = T(n-2)+2+2-\frac{1}{n}-\frac{1}{n-1} = \cdots = T(0)+2n-\sum_{k=1}^{n}\frac{1}{k} \asymp n.$$

4.4 Два алгоритъма

Ще започнем с два често срещани алгоритъма, които се възползват от това, че данните идват сортирани:

- двоично търсене;
- алгоритъма за задачата **2SUM**.

Нека започнем с алгоритъма за двоично търсене:

```
1  BinarySearch(A[1...n] ∈ array(ℤ); v ∈ ℤ):
2      l ← 1
3      r ← n
4
5      while l ≤ r:
6          m ← ⌊(l+r)/2⌋
7
8          if A[m] = v:
9              return m
10         else if A[m] < v:
11             l ← m + 1
12         else:
13             r ← m − 1
14
15     return −1
```

При подаден сортиран целочислен масив $A[1 \dots n]$ и цяло число v, функцията `BinarySearch`$(A[1 \dots n], v)$ ще върне индекс на $A[1 \dots n]$, в който се намира v, ако има такъв, иначе ще върне -1.

Инвариант 4.4.1. *При всяко достигане на проверката за край на цикъла на ред* 6 *имаме, че стойността* v *не се намира измежду двата масива* $A[1 \dots l-1]$ *и* $A[r+1 \dots n]$.

Доказателство.
База. При първото достигане имаме, че $l = 1$ и $r = n$. Наистина v не се намира измежду двата масива $A[1 \dots 1-1]$ и $A[n+1 \dots n]$.
Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава v не се намира измежду $A[1 \dots l-1]$ и $A[r+1 \dots n]$, и понеже достигането е непоследно, $l \leq r$. Тогава $m = \left\lfloor \frac{l+r}{2} \right\rfloor$, откъдето $l \leq m \leq r$. Трябва да разгледаме следните три случая:

1 сл. $A[m] = v$ – това няма как да е изпълнено понеже достигането е нефинално;

2 сл. $A[m] < v$ – понеже $A[1 \dots n]$ е сортиран, няма как v да се намира измежду $A[1 \dots m]$, откъдето v не се намира измежду $A[1 \dots \underbrace{\texttt{m} + \texttt{1}}_{l_{new}} - 1]$ и $A[r+1 \dots n]$;

3 сл. $A[m] > v$ – напълно дуален на 2 сл.

Терминация. От цикъла винаги ще излезем, защото или ще открием v, или величината $r - l$ ще намалява, докато не стане отрицателна. Излизането става по два начина:

- Ако не е изпълнено условието на ред 6 т.е. $l > r$, то тогава $l - 1 \geq r$ и понеже v не се намира измежду $A[1 \dots l-1]$ и $A[r+1 \dots n]$, v не се намира във $A[1 \dots n]$. Накрая алгоритъмът ще върне -1, което наистина е желания резултат.
- Ако е изпълнено е условието на ред 9 т.е. $A[m] = v$, то тогава алгоритъмът коректно връща m.

□

Сега ще разгледаме алгоритъм за задачата **2SUM**.
Ще искаме алгоритъм, който при подаден сортиран целочислен масив $A[1 \dots n]$ и цяло число t разпознава дали има $1 \leq i < j \leq n$, за които:

$$A[i] + A[j] = t.$$

Такива двойки $(A[i], A[j])$ ще наричаме *двади*.

Алгоритъмът е следния:

```
1  Solve2SUM(A[1...n] ∈ array(ℤ);  t ∈ ℤ):
2      (l, r) ← (1, n)
3
4      while l < r:
5          if A[l] + A[r] = t:  return 𝕋
6          if A[l] + A[r] < t:  l ← l + 1
7          if A[l] + A[r] > t:  r ← r − 1
8
9      return 𝔽
```

Инвариант 4.4.2. *При всяко достигане на проверката за край на цикъла на ред* 5*, всички диади са в* $A[l \dots r]$.

Доказателство.
База. При първото достигане имаме, че $l = 1$ и $r = n$. Тогава твърдението е тривиално изпълнено.
Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава всички диади са в $A[l \dots r]$ и $l < r$. Разглеждаме три случая:

1 сл. $A[l] + A[r] = t$ – това няма как да е изпълнено понеже достигането е нефинално;

2 сл. $A[l]+A[r] < t$ – тогава понеже $A[1 \dots n]$ е сортиран, за всяко $l \leq i \leq r$ имаме, че $A[l]+A[i] \leq A[l]+A[r] < t$, което означава, че l не участва в никоя диада във $A[l \dots r]$, следователно всички диади се намират в $A[\underbrace{l+1}_{l_{new}} \dots r]$;

3 сл. $A[l] + A[r] > t$ – напълно дуален на 2 сл.

Терминация. От цикъла винаги ще излезем, защото или ще открием v, или величината $r-l$ ще намалява, докато не стане 0. Излизането става по два начина:

- Ако не е изпълнено условието на ред 4 т.е. $l \geq r$, то тогава няма диади в $A[1 \dots n]$, и алгоритъмът коректно ще върне $\mathbb{F}$.
- Ако е изпълнено условието на ред 5 т.е. $A[l] + A[r] = t$, то тогава алгоритъмът връща $\mathbb{T}$ т.е. точно това, което искаме.

□

Двата алгоритъма са доста подобни, използват една често срещана техника за търсене в дадено множество от елементи. Търсенето започва с цялото множество и то постепенно се смалява. Разбира се, тук разгледаните алгоритми имат разлика в сложността, поради разликата в стесняването:

- първият алгоритъм има сложност $O(\log(n))$, понеже разликата между r и l винаги намалява двойно;
- вторият алгоритъм има сложност $O(n)$, понеже разликата между r и l винаги намалява с единица.

4.5 Задачи

Задача 4.1. Да се напише колкото се може по-бърз алгоритъм, който приема масив от различни цели числа $A[1 \dots n]$ с $n \geq 3$, за който има $1 < i < n$ такова, че $A[1 \dots i]$ е сортиран възходящо и $A[i \dots n]$ е сортиран низходящо, и връща това i. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.2. Да се напише колкото се може по-бърз алгоритъм, който при подаден сортиран целочислен масив $A[1 \dots n]$ и число t, което се намира в масива, връща най-малкият и най-големият индекс, на които t се намира в $A[1 \dots n]$. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.3. Да се напише колкото се може по-бърз алгоритъм, който при подаден сортиран целочислен масив $A[1 \dots n]$ и числа $k \geq 2$ и t, връща дали има $1 \leq i_1 < i_2 < \dots < i_k \leq n$, за които:

$$A[i_1] + A[i_2] + \dots + A[i_k] = t.$$

След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.4. За $a, b, x \in \mathbb{Z}$ казваме, че a е по-близо от b до x, ако:

$$|a - x| < |b - x| \vee (|a - x| = |b - x| \,\&\, a < b).$$

Да се напише колкото се може по-бърз алгоритъм, който приема целочислен масив $A[1 \dots n]$, и връща най-близките k на брой числа до t в $A[1 \dots n]$. След това да се докаже неговата коректност, и да се изследва сложността му по време. Може ли да се напише по-бърз алгоритъм при предположение че $A[1 \dots n]$ е сортиран?

Задача 4.5. Да се напише колкото се може по-бърз алгоритъм, който приема масив $A[1 \dots n]$, съставен от числата $0, 1$ и 2, и го сортира. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.6. Да се напише колкото се може по-бърз алгоритъм, който приема масив $I[1 \dots n]$, съставен от двойки числа, които представят някакъв затворен интервал от цели числа ($[a, b]$ за някои $a, b \in \mathbb{Z}$), и връща нов масив, в който са слети всички интервали с непразно сечение. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.7. Да се напише колкото се може по-бърз алгоритъм, който приема целочислен масив $A[1 \dots n]$, и връща нов масив от квадратите на $A[1 \dots n]$, който е сортиран. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.8. Да се напише колкото се може по-бърз алгоритъм, който приема целочислен масив $A[1 \dots 2n]$, и връща:

$$\min\{\max\{A'[2i] + A'[2i-1] \mid 1 \leq i \leq n\} \mid A'[1 \dots 2n] \text{ е пермутация на } A[1 \dots 2n]\}.$$

След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.9. Да се напише колкото се може по-бърз алгоритъм, който приема два целочислени масива $A[1 \dots n]$ и $B[1 \dots n]$, и връща:

$$\min\{\sum_{i=1}^{n} |A'[i] - B'[i]| \mid A'[1 \dots n] \text{ е пермутация на } A[1 \dots n] \text{ и } B'[1 \dots n] \text{ е пермутация на } B[1 \dots n]\}.$$

След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4.10. Да се напише колкото се може по-бърз алгоритъм, който приема целочислен масив $A[1 \dots n]$ и го подрежда така, че всички отрицателни числа да са вляво от всички неотрицателни. След това да се докаже неговата коректност и да се изследва сложността му по време и памет.

Задача 4.11. Статиите на даден учен са номерирани от 1 до n. Индексът на Хирш или h-индексът за този учен ще наричаме най-голямото число k, за което поне k негови статии са цитирани поне k пъти. Да се състави алгоритъм, който приема масив $C[1 \dots n]$ от броя на цитиранията на статиите на този учен и връща неговият h-индекс.

Глава 5

Алгоритми върху графи

5.1 Защо изобщо се занимаваме с графи?

Графите са може би най-приложимата структура в областта на компютърните науки. Тяхната моделираща мощ е несравнима с тази на останалите структури. Те могат се използват за моделиране на:

- приятелски връзки в социални мрежи;
- пътни мрежи в навигационни системи;
- йерархични системи;
- биологични мрежи.

Някои от задачите, които могат да решават са:

- намиране на най-къс път от точка A до точка B;
- намиране на съвместима наредба на дадени задачи;
- намиране на най-добро разписание на полети;
- класифициране на уебсайтове по популярност;
- маркетинг в социални мрежи;
- валидация на текст.

5.2 Как представяме графите в паметта?

В зависимост от нашите цели графите могат да бъдат представени в паметта по различни начини. Най-използваните начини са:

- **Списък на съседство**:

  За всеки връх се пазят в списък съседите му (и теглата ако има такива).

- **Матрица на съседство**:

  Пази се булева (може и числова ако графът е тегловен) таблица със всевъзможните комбинации от двойки върхове. Ако между два върха има ребро, то в съответната клетка пише единица (или теглото на реброто при тегловен граф), иначе нула.

- **Списък с ребрата**:

  Множеството от ребра идва като списък. Обикновено ако графът е неориентиран се пази само една пермутация на реброто.

Матрицата на съседство се използва по-рядко. Този подход е добър, когато графите са гъсти т.е. има много ребра в тях. В противен случай ние заемаме много повече памет от колкото ни е нужна. За сметка на това можем да проверим дали между два върха има ребро за константно време.
Списъците на съседство са по-пестеливи от към памет в средния случай, обаче за сметка на това по-бавно се проверява съседство между два върха. Този подход е добър, когато графите са редки т.е. имат малко ребра в тях. Също така ако по някаква причина ни трябва да изброяваме точно съседите на някакъв връх (да кажем за някакво обхождане), това очевидно е най-добрият начин. В най-лошия случай заемаме двойно повече памет от подхода с матрицата.
Списъка с ребрата е най-пестеливия начин от тези три. Пази се минималното количество нужна информация. Проблемът тук е, че проверката за съседство и изброяването на съседи на даден връх са бавни. Обаче това представяне все пак се използва, например когато искаме да построим МПД. Накратко, сложностите са такива:

подход	памет	$(u, v) \in E$	изброяване на съседите
списък на съседство	$O(\|V\| + \|E\|)$	$O(\|V\|)$	$\Theta(\|N(v)\|)$
матрица на съседство	$\Theta(\|V\|^2)$	$\Theta(1)$	$\Theta(\|V\|)$
списък с ребра	$\Theta(\|E\|)$	$O(\|E\|)$	$\Theta(\|E\|)$

5.3 Код на алгоритмите за обхождане на графи

Първият алгоритъм, за обхождане в широчина, е "по-предпазлив". Той обхожда върховете на слоеве, започвайки с някакъв първоначален връх на слой 0. Намирайки се в слой k, ако преминем с ребро до необходим връх, ще се озовем в слой $k+1$:

```
1  HelperBFS(G = (V, E) ∈ 𝔊raph;  s ∈ V;
2              vis ∈ array(𝔹);  r ∈ dynamic_array(V)):
3      init empty q ∈ queue(V)
4      q.push(s)
5      vis[s] ← 𝕋
6
7      while ¬q.empty():
8          u ← q.pop()
9          r.push(u)
10
11         for v ∈ adj_G(u):
12             if ¬vis[v]:
13                 vis[v] ← 𝕋
14                 q.push(v)
15
16 BFS(G = (V, E) ∈ 𝔊raph where V = {1, ..., n}):
17     vis[1...n] = [𝔽, ..., 𝔽]
18     init empty r ∈ dynamic_array(V)
19
20     for i ← 1 to n:
21         if ¬vis[i]:
22             HelperBFS(G, i, vis, r)
23
24     return r
```

Сложност на алгоритъма за търсене в широчина в най-лошия случай:

- по време – $\Theta(|V| + |E|)$;
- по памет – $\Theta(|E|)$.

Тук виждаме силата на представянето чрез списъци на съседство. Ако например тук бяхме използвали матрица на съседство, алгоритъмът ни винаги щеше да има сложност $\Theta(|V|^2)$.

Нека сега разгледаме другия алгоритъм – за обхождане в дълбочина. Тук гледаме да влизаме колкото се може "по-надълбоко" в даден връх т.е. избираме произволно ребро в текущ връх докато можем, и след това се връщаме на предния и правим същото:

```
1  HelperDFS(G = (V, E) ∈ Graph;  u ∈ V;
2                vis ∈ array(𝔹);  r ∈ dynamic_array(V)):
3      r.push(u)
4      vis[u] ← 𝕋
5
6      for v ∈ adj_G(u):
7          if ¬vis[v]:
8              HelperDFS(G, v, vis, r)
9
10 DFS(G = (V, E) ∈ Graph where V = {1, ..., n}):
11     vis[1...n] ← [𝔽, ..., 𝔽]
12     init empty r ∈ dynamic_array(V)
13
14     for i ← 1 to n:
15         if ¬vis[i]:
16             HelperDFS(G, i, vis, r)
17
18     return r
```

Сложността по време и памет на алгоритъма за търсене в дълбочина е същата като на този за търсене в широчина.

5.4 Най-къси пътища в тегловен граф

Започваме с може би най-известния нетривиален графов алгоритъм – алгоритъмът на Дийкстра за намиране на най-къси пътища в тегловен граф. При него започваме с един стартов връх, и намираме най-късите пътища между този стартов връх и всеки достижим от него.
Ето как става това:

```
1  Dijkstra(G = (V, E, w) ∈ 𝔊𝔯𝔞𝔭𝔥(weighted) where V = {1, ..., n}; s ∈ V):
2      init empty q ∈ priority_queue_min(ℕ × V, <_lex(ℕ×V))
3      d[1...n] ← [−∞, ..., −∞]
4      π[1...n] ← [0, ..., 0]
5      d[s] = 0
6      q.push((d[s], s));
7
8      while ¬q.empty():
9          (d_u, u) ← q.pop()
10
11         for v ∈ adj_G(u):
12             if d_u + w(u, v) < d[v]:
13                 π[v] ← u
14                 d[v] ← d_u + w(u, v)
15                 q.push((d[v], v))
16
17     return π[1...n]
```

Алгоритъмът има сложност по време $\Theta((|V|+|E|)\log(|V|))$ в най-лошия случай.

5.5 Структурата Union-find/Disjoint-union

Ще разгледаме метод за поддържане на разбиване на множества от вида $\{1,\ldots,n\}$. Искаме да започнем от разбиването $\{\{1\},\ldots,\{n\}\}$, и след това бързо да можем да обединяваме множества и да проверяваме дали някои i,j попадат на едно и също място в разбиването. За да може тези проверки и сливания да стават бързо, се използва структурата Union-find (понякога се нарича Disjoint-union). Ще имаме единствена функция unify(i,j), която приема $i,j\in\{1,\ldots,n\}$ и слива множествата от разбиването, в които i и j участват. Ако преди това те са били в едно и също множество, то тогава накрая връщаме $\mathbb{F}$, иначе връщаме $\mathbb{T}$.

Имплементацията е следната:

```
1  struct UnionFind:
2      Constructor(n ∈ ℕ):
3          init l[1...n] ∈ dynamic_array({1,...,n})
4          init s[1...n] ∈ dynamic_array(ℕ)
5
6          for i ← 1 to n;
7              l[i] ← i
8              s[i] ← 1
9
10     GetLeader(x ∈ {1,...,n}):
11         if x ≠ l[x]:
12             l[x] ← GetLeader(l[x])
13
14         return x
15
16     Unify(x, y ∈ {1,...,n}):
17         x ← GetLeader(x)
18         y ← GetLeader(y)
19
20         if x = y:
21             return 𝔽
22
23         if s[x] < s[y]:
24             swap(x, y)
25
26         l[y] ← x
27         s[x] ← s[x] + s[y]
28
29         return 𝕋
```

Сложността на извикването на unify е $O(\alpha(n))$, където α е обратната функция на Акерман. Тази функция расте изключително бавно – $\alpha(n) \leq 4$ за $n < 10^{600}$.

5.6 Алгоритъмът на Крускал за намиране на МПД

Алгоритъмът на Крускал работи по много естествен начин. Стараем се да приоритизираме ребрата с най-малки тегла. Не да ги добавяме само ако биха образували цикъл.
Нека сега формализираме разсъжденията си. Даден тегловният граф $G = (V, E, w)$. Него ще пазим като масив от ребра $T[1 \dots k] \in \text{arr}(E)$. Сега дефинираме релацията $\leq_G \subseteq E \times E$ така:

$$e_1 \leq_G e_2 \overset{def}{\iff} w(e_1) \leq w(e_2).$$

Вече можем да преминем на имплементацията:

```
1  Kruskal(E[1...m] where E[i] = (u,v) for some u,v ∈ {1,...,n}):
2      Sort(E[1...m], ≤_G)
3      init empty t ∈ dynamic_array(type_of_element(E[1...m]))
4      init dsu ∈ UnionFind(n)
5
6      for i ← 1 to m:
7          (u,v) ← E[i]
8
9          if dsu.Unify(u, v):
10             t.push((u,v))
11
12     return t
```

Сложността на алгоритъма по време е $\Theta(|E| \log(|E|))$ в най-лошия случай.

5.7 Задачи

Задача 5.1. Да се напише алгоритъм, който при подаден граф намира броя на свързаните компоненти.

Задача 5.2. Да се напише алгоритъм, който при подаден граф проверява дали той е цикличен.

Задача 5.3. Да се напише алгоритъм, който при подаден граф проверява дали той е дърво.

Задача 5.4. Да се напише алгоритъм, който при подаден (може и ориентиран) граф и два негови върха намира най-късия път между тях.

Задача 5.5. Да се напише алгоритъм, който при подаден (може и ориентиран) граф, два негови върха и дължина намира броят на пътища между тях със съответната дължина.

Задача 5.6. Да се напише алгоритъм, който при подаден граф проверява дали той е двуделен.

Задача 5.7. Трябва да изпълним задачи $1, \ldots, n$. Обаче имаме допълнителни изисквания $R[1 \ldots k]$ от вида $\langle i, j \rangle$, които казват *"задача i трябва да се изпълни преди задача j"*. Да се напише алгоритъм, който при подадени изисквания намира последователност от задачи, която удовлетворява тези изисквания. Ако няма такива, да се върне съобщение за грешка.

Задача 5.8. В някакъв град има n души с етикети от 0 до $n - 1$. Има слухове, че един от тези хора тайно е съдията на града. Ако такъв човек има, то тогава:

- съдията не вярва на никого;
- всеки вярва на съдията, освен самия него.

Масив на вярата за този град ще наричаме всеки масив $T[1 \ldots k]$ от наредени двойки $\langle i, j \rangle$ (за $0 \leq i, j < n$), които казват *"човекът с етикет i вярва на човека с етикет j"*. Да се напише алгоритъм, който при подаден масив на вярата, връща етикета на съдията. Ако няма съдия, да се върне -1.

Задача 5.9. Да се напише алгоритъм, който при подаден ориентиран тегловен граф намира теглата на минималните пътища между всеки два върха.

Задача 5.10. Да се напише алгоритъм, който при подаден тегловен ориентиран ацикличен граф и негов начален връх намира най-късите пътища от този начален връх до всички достижими от него.

Задача 5.11. Да се напише алгоритъм, който при подаден свързан цикличен граф с ребро, чието премахване ще превърне графа в дърво, връща това ребро.

Задача 5.12. Нека е дадено крайно множество от променливи $X = \{x_1, \ldots x_k\}$. Уравнения над X ще наричаме всички низове от вида $x_i = x_j$ и $x_i \neq x_j$ за някои $1 \leq i, j \leq k$. Да се напише алгоритъм, който при подадено множество от променливи X и списък $E[1 \ldots n]$ от уравнения над X проверява дали системата, съставена от списъка с уравнения има решение.

Задача 5.13. Алис и Боб имат граф с върхове измежду 0 и n и три типа ребра:

- ребрата от тип 1 могат да бъдат траверсирани само от Алис;
- ребрата от тип 2 могат да бъдат траверсирани само от Боб;
- ребрата от тип 3 могат да бъдат траверсирани и от двамата.

Да се напише алгоритъм, който при подаден масив от ребра $E[1 \dots k]$ т.е. тройки от типа $\langle type, i, j \rangle$ за някои $type \in \{1, 2, 3\}$ и $i, j \in \{1, \dots, n\}$ връща максималния брой ребра, които могат да се махнат, и графът пак да може да бъде обходен от Алис и Боб. Ако някой от двамата не може да обходи първоначалния граф, да се върне -1.

Задача 5.14. За всеки две точки $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ в $\mathbb{Z} \times \mathbb{Z}$, цената на свързване на p_1 и p_2 ще бъде $|x_1 - x_2| + |y_1 - y_2|$. Да се напише алгоритъм, който при подаден масив $P[1 \dots n]$ от точки намира минималната обща цена на свързване, за която между всеки две точки от $P[1 \dots n]$ ще има път.

Глава 6

Динамично програмиране

6.1 Какво е динамично програмиране?

Динамичното програмиране не е нито динамично, нито програмиране. Това е както оптимизационен метод, така и алгоритмична парадигма, която е разработена от Ричард Белман през 50-те години на миналия век. В този метод една зачача се разделя на подзадачи по рекурсивен начин. Той се използва в два случая:

- при задачи, които имат припокриваща се подструктура т.е. задачата се разделя на подзадачи, които се срещат няколко пъти;
- при задачи, които имат оптимална подструктура т.е. оптимално решение може да се конструира от оптимални решения на подзадачите.

6.2 Прости примери за динамично програмиране

Да кажем, че искаме да сметнем n-тото число на Фибоначи. Един начин е да караме по рекурентното уравнение:

```
1  Fib(n ∈ ℕ):
2      if n < 2:
3          return n
4
5      return Fib(n − 1) + Fib(n − 2)
```

Проблемът е, че получаваме експоненциална сложност по време. Нещо, което можем да направим, е да пазим вече пресметнатите стойности, за да не се налага да ги пресмятаме пак:

```
1  FibDP(n ∈ ℕ):
2      if n < 2:
3          return n
4
5      declare F[0...n] ∈ array(ℤ)
6
7      F[0] ← 0
8      F[1] ← 1
9
10
11     for i ← 2 to n:
12         F[i] ← F[i − 1] + F[i − 2]
13
14     return F[n]
```

Това е пример за задача с припокриваща се подструктура, с решение по схемата **динамично програмиране**. Успяхме да решим задачата за линейно време.
Нека сега видим пример за задача с оптимална подструктура. Да кажем, че имаме два низа $S_1[1 \dots n]$ и $S_2[1 \dots m]$ и искаме да пресметнем дължината на най-дългата обща подредица на S_1 и S_2. Лесно се вижда, че дължината $\mathrm{LCS}_{S_1,S_2}(i,j)$ на най-дългата подредица на $S_1[1 \dots i]$ и $S_2[1 \dots j]$ може да се пресметне рекурсивно така:

$$\mathrm{LCS}_{S_1,S_2}(i,j) = \begin{cases} 0 & \text{, ако } i = 0 \text{ или } j = 0 \\ \mathrm{LCS}_{S_1,S_2}(i-1,j-1) + 1 & \text{, ако } i,j > 0 \text{ и } S_1[i] = S_2[j] \\ \max\{\mathrm{LCS}_{S_1,S_2}(i-1,j), \mathrm{LCS}_{S_1,S_2}(i,j-1)\} & \text{, ако } i,j > 0 \text{ и } S_1[i] \neq S_2[j] \end{cases}$$

Ако искаме да пресметнем това със обикновена рекурсия, отново ще получим експоненциална сложност по време. Отново можем да направим решение по схемата **динамично програмиране** със сложност $\Theta(n \cdot m)$. Единственото, което трябва да се направи, е да се измисли последователност за пресмятане на:

$$\begin{pmatrix} \mathrm{LCS}_{S_1,S_2}(0,0) & \dots & \mathrm{LCS}_{S_1,S_2}(n,0) \\ \vdots & \ddots & \vdots \\ \mathrm{LCS}_{S_1,S_2}(m,0) & \dots & \mathrm{LCS}_{S_1,S_2}(n,m) \end{pmatrix}$$

Важно е да искаме преди пресмятането на всяка клетка, необходимите за нея данни вече са готови. Ето как може да стане това:

```
1  LCS(S_1[1...n], S_2[1...m] ∈ string):
2      declare DP[0...n][0...m] ∈ matrix(ℕ)
3
4      for i ← 0 to n:
5          DP[i][0] ← 0
6
7      for j ← 0 to m:
8          DP[0][j] ← 0
9
10     for i ← 1 to n:
11         for j ← 1 to m:
12             if S_1[i − 1] = S_2[j − 1]:
13                 DP[i][j] ← DP[i − 1][j − 1] + 1
14             else:
15                 DP[i][j] ← max(DP[i − 1][j], DP[i][j − 1])
16
17     return DP[n][m]
```

6.3 Динамично програмиране за решаване на комбинаторни задачи

Вижда се, че тази техника е много удобна за бързо пресмятане на рекурентни зависимости. Едно приложение е в решаването на комбинаторни задачи. Да кажем, че искаме да пресметнем бързо $\binom{n}{k}$.
Нека първо си припомним дефиницията:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Ние ще пресметнем $\binom{n}{k}$ точно по този начин. За тази цел единственото нещо, което се иска да сметнем трите стойности – $n!, k!$ и $(n-k)!$.

Това става елементарно по следния начин:

```
1  BinomialFactorial(n, k ∈ ℕ):
2      if n < k:
3          return 0
4
5      declare F[0...n] ∈ array(ℤ)
6      F[0] ← 1
7
8      for i ← 1 to n:
9          F[i] ← i · F[i − 1]
10
11     return F[n]/(F[k] · F[n − k])
```

Получихме сложност по време $\Theta(n)$ т.е. имаме сравнително бързо решение. Обаче то не е практично. Проблемът е, че функцията $n!$ расте много бързо. Ние ще работим с големи стойности във $F[0 \dots n]$, но крайният отговор ще е много по-малък от това. Ако искаме да си гарантираме възможно най-малки стойности по време на изчисление, трябва да го пресметнем за време $\Theta(n \cdot k)$ чрез формулата:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Нека сега се опитаме да пресметнем броят T_n[*] на двоични дървета за търсене с n различни върха. При $n = 0$ положението е ясно. Ако $n \geq 1$, то имаме няколко случая в зависимост от това кой връх е корен на дървото. Броят на двоичните дървета за търсене с корен i-тия по големина връх, където $1 \leq i \leq n$, е равен на броя двоичните дървета с $(i-1)$-те по-малки от него върха, умножен по броя на двоичните дървета с останалите $n - i$ върха. Това е точно $T_{i-1} \cdot T_{n-i}$. Така получаваме следното рекурентно уравнение:

$$T_0 = 1$$
$$T_n = \sum_{i=1}^{n} T_{i-1} \cdot T_{n-i} \text{ за } n > 0$$

Ясно е, че не искаме да пресмятаме T_n чрез рекурсия – това би било кошмарно бавно.

[*]Тези числа се наричат числа на Каталан.

Отново ще помним предишни изчисления, за да си забързаме алгоритъма до такъв със сложност $\Theta(n^2)$:

```
1  C(n ∈ ℕ):
2      declare C[0...n] ∈ array(ℤ)
3      C[0] ← 1
4
5      for i ← 1 to n:
6          C[i] ← 0
7
8          for j ← 1 to i:
9              C[i] ← C[i] + C[j − 1] · C[i − j]
10
11     return C[n]
```

6.4 Два интересни примера

За първия пример, нека имаме някакъв краен речник D от непразни думи над $\Sigma = \{a, \ldots, z\}$. Ще искаме при подаден низ над Σ да видим дали той може да се разбие на думи от речника (с възможни повторения). Нека например да вземем речника $D = \{\text{mango}, \text{i}, \text{icecream}, \text{like}, \text{with}\}$. Тогава думата "ilikeicecreamwithmango" може да се разбие на "i like icecream with mango".

Нека е подаден един низ $\alpha \in \Sigma^*$. Ако $\alpha = \varepsilon$, то тогава очевидно можем да получим α чрез конкатенация на думи от D. Ако $\alpha \neq \varepsilon$ и α се получава чрез конкатенация на думи от D, то тогава има $\beta \in \Sigma^*$ и $\gamma \in D$, за които е изпълнено, че $\alpha = \beta\gamma$ и β може да се получи чрез думи от D. Но $\gamma \neq \varepsilon$, т.е. успяхме да сведем задачата за α до задача за β, като $|\beta| < |\alpha|$. Нека сега да формализираме тези разсъждения. Нека $\alpha = \alpha_1 \ldots \alpha_n$, където $\alpha_i \in \Sigma$. Тогава булевата функция $\text{WB}_{D,\alpha}(i)$, която казва дали $\alpha_1 \ldots \alpha_i$ може да се разбие на думи от D, изглежда така:

$$\text{WB}_{D,\alpha}(i) = \begin{cases} \mathbb{T} & \text{, ако } i = 0 \\ \bigvee\limits_{\beta \in D} (\alpha_{i-|\beta|+1} \ldots \alpha_i = \beta \;\&\; \text{WB}_{D,\alpha}(i - |\beta|)) & \text{, иначе} \end{cases}$$

На нас това, което ни трябва, е $\mathrm{WB}_{D,\alpha}(|\alpha|)$. С малко мислене върху това как се пресмята $\mathrm{WB}_{D,\alpha}$, човек може да стигне до следното итеративно решение:

```
1  WordBreak(α ∈ Σ*;  D ⊆_fin Σ+):
2      WB[0...|α|] ← [𝔽,...𝔽]
3      WB[0] ← 𝕋
4
5      for i ← 1 to |α|:
6          for β ∈ D:
7              if i < |β|:
8                  continue
9
10             if (i = |β| ∨ WB[i − |β|]):
11                 if α_{i−|β|+1}...α_i = β:
12                     WB[i] ← 𝕋
13                     break
14
15     return WB[|α|]
```

Да направим сега малко анализ. Ако $|\alpha| = n, |D| = m$, и $\max\{|\beta| \mid \beta \in D\} = k$, то това решение има сложност по време $O(n \cdot m \cdot k)$, понеже за всяко $1 \leq i \leq n$ и за всяка дума $\beta \in D$ (те са m на брой) проверяваме дали $\alpha_{i-|\beta|+1} \ldots \alpha_i = \beta$, което става за време $O(k)$, и дали $\mathrm{WB}_{D,\alpha}(i - |\beta|) = \mathbb{T}$, което поради наличието на масива $WB[0 \ldots n]$ става за константно време.
Този масив доста помага, ако не го бяхме ползвали, сложността щеше да се опише (горе долу) с рекурентното уравнение:

$$\begin{aligned} T(n) = \sum_{i=1}^{n}(m \cdot k \cdot T(n-i)) + \Theta(1) &= m \cdot k \cdot \sum_{i=0}^{n-1} T(i) + \Theta(1) \\ &= m \cdot k \cdot T(n-1) + m \cdot k \cdot \sum_{i=0}^{n-2} T(i) + \Theta(1) \\ &= m \cdot k \cdot T(n-1) + T(n-1) \\ &= (m \cdot k + 1)T(n-1). \end{aligned}$$

В този случай $T(n) \asymp (m \cdot k + 1)^n$.
В горния алгоритъм сложността по памет очевидно е $\Theta(n)$, заради допълнителния масив, който заделяме.

Вторият пример ще бъде под формата на игра. Наредени са n монети със стойности съответно $v_1, \ldots, v_n$. Редуваме се с опонент да теглим една монета от избран от краищата на редицата, докато монетите не свършат. Накрая всеки човек печели толкова, колкото е изтеглил. В случай че играем първи, каква печалба можем да си гарантираме?
Първо да започнем с двата най-прости типа игри – с една или две монети. В играта с една монета е ясно, че най-голямата печалба, която можем да си гарантираме, е стойността на монетата. В игра с две монети със стойности съответно v_1, v_2, най-голямата гарантирана печалба е очевидно $\max\{v_1, v_2\}$.
Нека сега в общия случай имаме следната партия:

Ще се опитаме да сведем тази по-сложна партия, до няколко по-прости. Имаме две възможности:

1. Ако изберем първата монета, опонента ще трябва да избира в конфигурацията:

   Той също има две възможности:

   (а) Ако опонента избере първата монета, ни оставя в конфигурацията:

   Тук можем да си мислим, че ние сме си заделили на страна печалба v_1, опонента си е заделил на страна печалба v_2, и играта започва наново в горната конфигурация.

   (б) Ако пък избере последната монета, ни оставя в конфигурацията:

   Тук можем да си мислим, че ние сме си заделили на страна печалба v_1, опонента си е заделил на страна печалба v_n, и играта започва наново в горната конфигурация.

2. Ако изберем последната монета, опонента ще трябва да избира в конфигурацията:

Той също има две възможности:

(а) Ако опонента избере първата монета, ни оставя в конфигурацията:

Тук можем да си мислим, че ние сме си заделили на страна печалба v_n, опонента си е заделил на страна печалба v_1, и играта започва наново в горната конфигурация.

(б) Ако пък избере последната монета, ни оставя в конфигурацията:

Тук можем да си мислим, че ние сме си заделили на страна печалба v_n, опонента си е заделил на страна печалба v_{n-1}, и играта започва наново в горната конфигурация.

Нека $\mathrm{MP}(i,j)$ е максималната печалба, която може да се спечели от първия играч (в първоначалната игра), ако може да събира монети със стойности съответно $v_i, \ldots, v_j$. По предните разсъждения можем да пресметнем тази печалба рекурсивно така:

$$\mathrm{MP}(i,j) = \begin{cases} v_i & \text{, ако } i = j \\ \max\{v_i, v_j\} & \text{, ако } i = j+1 \\ \max\{v_i + \min\{\mathrm{MP}(i+2,j), \mathrm{MP}(i+1,j-1)\}, & \\ \qquad v_j + \min\{\mathrm{MP}(i,j-2), \mathrm{MP}(i+1,j-1)\}\} & \text{, иначе} \end{cases}$$

Във хода на втория играч минимизираме, защото той печели възможно най-много, когато ние печелим възможно най-малко. За задача на читателя оставяме да пресметне $\mathrm{MP}(1,n)$ итеративно.

6.5 Задачи

Задача 6.1. Да се напише колкото се може по-бърз алгоритъм, който пресмята $\binom{n}{k}$ с рекурентната формула от по-горе. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.2. Да се напише колкото се може по-бърз алгоритъм, който при подадено естествено число n, намира броят на начини човек да се изкачи по тях, като може да изкачва най-много 3 стъпала наведнъж. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.3. Да се напише колкото се може по-бърз алгоритъм, който при подадена булева матрица $A[1 \dots n, 1 \dots m]$ намира броя на пътищата (движейки се само надясно и надолу) от $(1, 1)$ до (n, m), които не минават през 1 в A. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.4. Да се напише колкото се може по-бърз алгоритъм, който при подадена матрица от естествени числа $A[1 \dots n, 1 \dots m]$ намира минималната цена на път (движейки се само надясно и надолу) от $(1, 1)$ до (n, m), като под цена на път разбираме сумата на всичките $A[i, j]$, срещнати по пътя. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.5. Да се напише колкото се може по-бърз алгоритъм, който при подаден целочислен масив $A[1 \dots n]$ намира дължината на най-дългата строго растяща негова подредица. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.6. Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от естествени числа $A[1 \dots n]$ и естествено число s намира броя на начините, по които могат да се изберат елементи на A, така че да сумата им да е s. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.7. За масив от положителни числа $A[1 \dots n]$ и $1 \leq i < j \leq n$ казваме, че можем да стигнем от i до j в A за една стъпка, ако и $j - i \leq A[i]$. Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от положителни числа $A[1 \dots n]$ намира минималния брой стъпки, с който можем да стигнем от 1 до n в A. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.8. Да се напише колкото се може по-бърз алгоритъм, който при подадено n и число k пресмята броят на начините да се стигне до k чрез хвърляния на зар с n страни, на които пише числата от 1 до n. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.9. Формула ще наричаме всеки низ от вида $B_0\sigma_1 B_1\sigma_2 B_2 \dots B_{n-1}\sigma_n B_n$, където $B_i \in \{\mathbb{T}, \mathbb{F}\}$ и $\sigma_i \in \{\vee, \wedge, \oplus\}$. Например низът $\mathbb{T} \wedge \mathbb{T} \oplus \mathbb{F} \vee \mathbb{T}$ е формула. В зависимост от това как слагаме скобите, оценката на израза може да е различна. Например изразът $(\mathbb{T} \wedge (\mathbb{T} \oplus (\mathbb{F} \vee \mathbb{T})))$ се остойностява като $\mathbb{F}$, докато изразът $(\mathbb{T} \wedge ((\mathbb{T} \oplus \mathbb{F}) \vee \mathbb{T}))$ се остойностява като $\mathbb{T}$, въпреки че и двата израза се получават от примерната формула. Да се напише колкото се може по-бърз алгоритъм, който при подадена формула да върне броят на различни скобувания, за които съответния израз се остойностява като $\mathbb{T}$. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.10. Имаме професионален крадец, който иска да ограби къщите в дадена улица. Проблемът е, че ако той ограби две съседни къщи, алармата ще се активира и полицията ще дойде. Той не иска това, защото в такъв случай няма да спечели нищо. Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от естествени числа $L[1 \dots n]$, където $L[i]$ е печалбата от къща i, връща максималната печалба на крадеца. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.11. Да се напише колкото се може по-бърз алгоритъм, който при подадена булева матрица $M[1 \dots n, 1 \dots m]$ намира най-голямото квадратче в M, съставено от 1, и връща неговото лице. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.12. Нека $n, l, r \in \mathbb{N}$ и $l \leq r$. Един целочислен масив $A[1 \dots n]$ ще наричаме (n, l, r)-интересен, ако:

- $l \leq A[i] \leq r$ за всяко $1 \leq i \leq n$;
- $\sum_{i=1}^{n} A[i] \equiv 0 \pmod 3$.

Да се напише колкото се може по-бърз алгоритъм, който при подадени $n, l, r \in \mathbb{N}$, за които $l \leq r$, връща броя на (n, l, r)-интересни масиви. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.13. Един целочислен масив $A[1 \dots n]$ наричаме аритметичен, ако $n \geq 3$ и за всяко $1 \leq i \leq n-2$ е изпълнено, че $A[i+2] - A[i+1] = A[i+1] - A[i]$. Да се напише колкото се може по-бърз алгоритъм, който при подаден целочислен масив $A[1 \dots n]$ намира броя на подредиците на A, които са аритметични масиви. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.14. Имаме n на брой къщи, които искаме да боядисаме със цветове c_1, c_2, c_3, като не може две съседни къщи да имат еднакъв цвят. Масив на цените ще наричаме всеки двумерен масив от положителни числа $P[1 \dots n, 1 \dots 3]$, където $P[i, j]$ ще бъде цената за боядисване на къща i със цвят c_j. Да се напише колкото се може по-бърз алгоритъм, който при подаден двумерен масив от положителни числа, намира минималната цена за боядисване на всички къщи. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6.15. Върху един масив от естествени числа $A[1 \dots n]$ можем да прилагаме следните две операции:

- да увеличим $A[i]$ с единица за някое $1 \leq i \leq n$;
- да намалим $A[i]$ с единица за някое $1 \leq i \leq n$.

Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от естествени числа $A[1 \dots n]$ връща минималния брой операции, които са нужни, за да стане масива монотонно ненамаляващ. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Глава 7

Долни граници

7.1 Какво са долни граници?

Дойде времето за по-депресиращите резултати в курса. До сега единственото, което правихме, беше да показваме, че за задача **X** може да се напише алгоритъм със времева сложност $O(f)$ или $\Theta(f)$ за някое $f \in \mathcal{F}$. Доста естествено изниква следния въпрос:

Възможно ли е да съществува по-бърз алгоритъм, който решава задачата **X***?*

Ясно е, че е неприемлив отговор от сорта на

Не мога да измисля такъв алгоритъм, следователно такъв алгоритъм не съществува.

В тази тема ще се опитаме да отговорим в някакъв смисъл положително на въпроси от този тип. Преди това нека въведем няколко дефиниции.
Изчислителна задача ще наричаме всяко множество от наредени двойки **X**, като:

- $\mathrm{Dom}(\mathbf{X})$ ще наричаме **вход**;
- $\mathrm{Rng}(\mathbf{X})$ ще наричаме **изход**.

За пример можем да вземем изчислителната задача **Sort**:

Вход: Целочислен масив $A[1 \dots n]$.
Изход: Пермутация $A'[1 \dots n]$ на $A[1 \dots n]$, за която $A'[1] \leq A'[2] \leq \cdots \leq A'[n]$.

Ще казваме, че **алгоритъм AlgX решава задачата X**, ако за всяко $x \in \mathrm{Dom}(\mathbf{X})$ е изпълнено $\langle x, \mathbf{AlgX}(x) \rangle \in \mathbf{X}$. Нека $\mathbf{X}$ е изчислителна задача и нека $f \in F$. Тогава:

- Казваме, че f е **долна граница** за $\mathbf{X}$, ако всеки алгоритъм, който решава $\mathbf{X}$, работи за време (или брой операции от конкретен вид) поне f^*.
- Казваме, че $\Omega(f)$ е **долна граница** за $\mathbf{X}$, ако всеки алгоритъм, който решава $\mathbf{X}$, работи за време (или брой операции от конкретен вид) $\Omega(f)$.

7.2 Техники за изследване на долни граници

Най-често се използват следните техники, които показват че задача $\mathbf{X}$ има долна граница за време f (или $\Omega(f)$):

- директни разсъждения за конкретния пример – тази техника обикновено се използва в малки задачи, където човек за сравнително малко време може да направи пълен анализ. Разбира се, тази техника се използва и при по-обобщените примери, но не толкова често.
- дърво на взимане на решения – тази техника се използва в задачи, където за решаването им се изисква задаването на редица въпроси, чиито отговор ни дава все повече и повече информация. Можем да си мислим за запитванията заедно с информацията, която носят, като едно дърво. Всеки въпрос ще разклонява дървото, докато накрая имаме цялата ни нужна информация в листата, и не трябва да задаваме повече въпроси. Тогава долната граница ще бъде височината на дървото.
- аргументация чрез противник – тази техника е трудна да се обясни без да се даде пример. Идеята е, че играем срещу противник, като нашата цел е да разкрием някаква информация, която уж е била предварително фиксирана. Противника обаче си измисля информацията на момента, като целта му е да ни накара да зададем колкото се може повече въпроси и в отговорите му на въпросите да няма противоречия.

*Тук се има предвид, че ако $T(n)$ е броят на стъпки (или операции от конкретен вид), за който алгоритъма завършва при вход с големина n, то $f(n) \leq T(n)$.

- чрез редукция[†] – ако знаем, че можем алгоритмично да сведем задача **Y** до задача **X** за време по-малко f и **Y** има долна граница за време f (или $\Omega(f)$), то тогава второто е изпълнено и за задача **X**. В някакъв смисъл тази редукция казва, че задачата **X** е поне толкова трудна, колкото задачата **Y**.

7.3 Техниките в действие

Ще започнем със пример за аргументация с противник. Решаваме задачата за максимален елемент – даден ни е като вход целочислен масив $A[1 \dots n]$ и искаме да получим като изход $\max\{A[i] \mid 1 \leq i \leq n\}$. Твърдим, че всеки алгоритъм, който решава задачата, използва поне $n-1$ сравнения. Ако при работа на алгоритъма е проверено условието "$A[i] \leq A[j]$" и е върнало $\mathbb{T}$, ще казваме, че $A[i]$ е загубило това сравнение и $A[j]$ е спечелило това сравнение. Нека $A[i]$ е максималният елемент на $A[1 \dots n]$. Тогава за всяко $j \neq i$ имаме, че $A[j]$ е загубило сравнение. Ако има $A[j]$, което не е загубило сравнение, то тогава $A[j]$ не е сравнявано с $A[i]$. Ако сменим $A[j]$ със $A[i]+1$, то тогава при изпълнението на алгоритъма резултатите (от сравненията) няма да се променят, и алгоритъмът ще върне $A[i]$, което е абсурд. Тъй като при всяко сравнение един връх печели, а другия губи, то за да постигнем тези $n-1$ загуби ни трябват $n-1$ сравнения. Ако си представим какво прави стандартния алгоритъм за намиране на максимум, той постоянно поддържа победител. Започва с един елемент, и когато той загуби, го заменя с друг. Накрая ще сме завършили с елемент, който никога не е губил, и е транзитивно е победил всички останали.
Нека сега дадем пример за дърво на вземане на решения. Разглеждаме задачата **Sort**. Ще покажем, че всеки сортиращ алгоритъм, който работи на базата на директни сравнения, работи за време $\Omega(n \log(n))$. Нека фиксираме $n \in \mathbb{N}$ и някакъв символ a. Нека $\mathcal{T}_n$ е множеството от всички дървета, за които е изпълнено, че:

- върховете са от вида $(a_i < a_j, P)$, където $1 \leq i, j \leq n,\ P \subseteq S_n$[‡] и $|P| \geq 2$, или са от вида $\sigma \in S_n$;
- ако $n > 1$, то коренът е $(a_i < a_j, S_n)$ за някои $1 \leq i, j \leq n$, иначе е единственият елемент на S_n;

[†]Това е може би най-приложимата техника от всички. Тя се използва не само в контекста на сложност. Оказва се, че е много удобно човек да може да говори за това дали една задача е *"по-трудна"* от друга в контекста на разрешимост.

[‡]S_n е симетричната група за $\{1, \dots, n\}$.

- за всеки връх от вида $(a_i < a_j, P)$, има $1 \leq k_1, k_2, m_1, m_2 \leq n$, за които:
  - лявото му дете (стига да е добре дефинирано т.е. дясната компонента не е $\varnothing$) е наредената двойка $(a_{k_1} < a_{m_1}, \{\sigma \in P \mid \sigma(i) < \sigma(j)\})$ (или ако се получава само една пермутация, само тя), и
  - дясното му дете (стига да е добре дефинирано т.е. дясната компонента не е $\varnothing$) е наредената двойка $(a_{k_2} < a_{m_2}, \{\sigma \in P \mid \sigma(i) \not< \sigma(j)\})$ (или ако се получава само една пермутация, само тя).

Тогава ако вземем произволен алгоритъм за сортиране **AlgX**, който е базиран на сравнение, при пресмятането на $\mathbf{AlgX}(A[1 \dots n])$, можем да забележим, че траверсираме някое дърво $T \in \mathcal{T}_n$ от корен до листо. В корена се намира първото запитване $a_i < a_j$ (т.е. можем да си мислим, че питаме дали $A[i] < A[j]$, където $A[i], A[j]$ са първоначалните стойности на входния масив), и спрямо отговора на дадено запитване ние се движим наляво или надясно в дървото. Накрая се намираме в листо $\sigma \in S_n$, която задава сортирана пермутация $A'[1 \dots n]$ на $A[1 \dots n]$ така – $A'[i] = A[\sigma(i)]$. Това означава, че за всяко сравнение по време на работа на $\mathbf{AlgX}(A[1 \dots n])$ можем да си мислим, че минаваме през едно ребро в T. В най-лошия случай входът $A[1 \dots n]$ би бил такъв, че да трябва да изминем максимален път от корен до листо т.е. път с дължина $h(T)$ (височината на дървото). Но T е двоично дърво с $n!$ листа и разклоненост 2, следователно $h(T) \geq \log_2(n!)$. Така в този случай извикването $\mathbf{AlgX}(A[1 \dots n])$ ще използва поне $\log(n!)$ сравнения. Тъй като $\log(n!) \asymp n \log(n)$, получаваме че всеки сортиращ алгоритъм, базиран на сравнения, прави $\Omega(n \log(n))$ сравнения. Нещо повече, това ще е вярно и за масив от естествени числа. Тук никъде не се възползваме от това, че числата са цели. Възползвахме се от това, че са безкрайно много.
Нека сега покажем един пример с редукция. Разглеждаме изчислителната задача **Матрьошки**:
Вход: Масив $T[1 \dots n]$ със елементи от вида (l, w, h) – дължините, широчините, височините на n играчки с форма на правоъгълен паралелепипед, които могат да се вложат една в друга.
Изход: Ред на влагане на играчките, като вътрешната играчка е първа.
Оказва се, че тази задача се решава (на базата на директни сравнения) за време $\Omega(n \log(n))$. Ще покажем това като сведем задачата за сортиране на естествени числа към **Матрьошки**. Нека **AlgM** е алгоритъм, който решава задачата **Матрьошки** със сложност по време $f(n)$.

Тогава следният алгоритъм очевидно сортира масива от естествени числа $A[1 \dots n]$:

1. Декларираме нов масив $T[1 \dots n]$.
2. За всяко $1 \leq i \leq n$ инициализираме $T[i]$ със $(A[i], A[i], A[i])$.
3. Извикваме $\mathbf{AlgM}(T[1 \dots n])$ с резултат $T'[1 \dots n]$.
4. Декларираме нов масив $A'[1 \dots n]$
5. За всяко $1 \leq i \leq n$ инициализираме $A'[i]$ със най-лявата компонента на $T'[i]$.
6. Връщаме $A'[1 \dots n]$.

Сложността на алгоритъма е $\Theta(n + f(n))$. Ако $f(n) = o(n \log(n))$, щяхме да получим сортиращ алгоритъм, който работи за време $o(n \log(n))$, което е абсурд.

7.4 Задачи

Задача 7.1. Един целочислен масив $A[1 \dots 2n]$ ще наричаме симетричен, ако за всяко $1 \leq i \leq n$ е изпълнено, че $A[i] = A[2n-i+1]$. Да се докаже, че сортирането на симетрични масиви чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.2. Един целочислен масив $A[1 \dots 2n]$ ще наричаме специален, ако за всяко $1 \leq i \leq n$ е изпълнено, че $A[2i] < A[2i-1]$. Да се докаже, че сортирането на специални масиви чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.3. Разглеждаме задачата **SortPointsCounterClockwise**:

Вход: Точки $P_1, \dots, P_n \in \mathbb{N} \times \mathbb{N}$.
Изход: Последователност $P'_1, \dots, P'_n$ на точките $P_1, \dots, P_n$, за която за всяко $1 \leq i < n$ е изпълнено, че най-малкото завъртане от вектора $\overrightarrow{OP'_i}$ към вектора $\overrightarrow{OP'_{i+1}}$ става обратно на часовниковата стрелка, където O е началото на координатната система.

Да се докаже, че решението на задачата **SortPointsCounterClockwise** чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.4. Разглеждаме задачата **SortPointsAngle**:

Вход: Точки $P_1, \ldots, P_n \in \mathbb{N} \times \mathbb{N}$.
Изход: Подреждане на точките $P_1, \ldots, P_n$ по големина на ъгъла, който сключва радиус-векторът с Ox.

Да се докаже, че решението на задачата **SortPointsAngle** чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.5. Нека $A[1 \ldots n]$ и $B[1 \ldots n]$ са сортирани целочислени масиви. Да се докаже, че $2n - 1$ е долна граница за броя на сравненията при сливането на тези два масива в един сортиран масив $C[1 \ldots 2n]$.

Задача 7.6. Даден е граф с $2n$ върха. Интересуват ни въпроси от вида:

"Има ли ребро от връх i до връх j?"

Да се докаже, че n^2 е долна граница за броя въпроси, който е нужен, за да установим дали дадения граф е свързан.

Задача 7.7. Искаме да познаем число между 1 и n, което някой друг човек е намислил. За целта можем да му задаваме въпроси (на които той трябва да отговори честно) от вида:

"Вярно ли е, че k е по-малко от намисленото число?"

Да се докаже, че $\lceil \log_2(n) \rceil$ е долна граница за броя въпроси, който е нужен, да познаем намисленото число.

Задача 7.8. Да се докаже, че сортирането на двоична пирамида чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.9. Дефинираме вълнист масив индуктивно:

- Всеки едноелементен масив е вълнист.
- Масив $A[1 \ldots n]$ (където $n > 1$) е вълнист, ако
  1. масивът $A[1 \ldots \lfloor \frac{n}{2} \rfloor]$ е сортиран, а
  2. масивът $A[\lfloor \frac{n}{2} \rfloor + 1 \ldots n]$ е вълнист.

Да се докаже, че пермутирането на масив до вълнист изисква време $\Omega(n \log(n))$.

Задача 7.10. Да се докаже, че строенето на двоична пирамида от масив $A[1 \ldots n]$ изисква $n - 1$ сравнения.

Задача 7.11. Разглеждаме задачата **ElementUniqueness**:

Вход: Целочислен масив $A[1 \dots n]$.
Въпрос: Вярно ли е, че масивът $A[1 \dots n]$ съдържа само уникални елементи?

Да се докаже, че решението на задачата **ElementUniqueness** чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.12. Разглеждаме задачата **Mode**:

Вход: Целочислен масив $A[1 \dots n]$.
Изход: Най-често срещано число в $A[1 \dots n]$.

Да се докаже, че решението на задачата **Mode** чрез сравнения изисква време $\Omega(n \log(n))$.

Задача 7.13. Разглеждаме задачата **ClosestPair**:

Вход: Целочислен масив $A[1 \dots n]$.
Изход: $\min\{|A[i] - A[j]| \mid 1 \leq i < j \leq n\}$.

Да се докаже, че решението на задачата **ClosestPair** чрез сравнения изисква време $\Omega(n \log(n))$.

Глава 8

Алгоритмична неподатливост

8.1 Класове на сложност P и NP

При решаването на различни видове задачи, които ни интересуват, е естествено да се опитаме да ги класифицираме по това колко са *"сложни"*. Това сме го виждали вече – в курса по ЕАИ сме класифицирали различни езици спрямо това каква машина може да решава въпроса за принадлежност към съответния език. Тук ще направим нещо подобно, разликата ще бъде в това, че ще се интересуваме от това за какво време се решава една задача.

- Класът на сложност **P** ще бъде множеството от всички изчислителни задачи за разпознаване, за които съществува алгоритъм с полиномиална времева сложност при най-лоши входни данни.
- Класът на сложност **NP** ще бъде множеството от всички изчислителни задачи за разпознаване, за които съществува алгоритъм с полиномиална времева сложност при всякакви входни данни, проверяващ отговора "ДА" на задачата с помощта на допълнителен параметър, наречен **сертификат**, който зависи от входните данни на задачата и чиято дължина е полиномиална спрямо тяхната.

Забележка. За класа **NP** има и алтернативна дефиниция – в нея не е нужен сертификат, но се допуска алгоритъмът да е недетерминиран. Идеята е, че той едновременно *"познава"* правилния отговор и го верифицира.

Нека дадем пример за сертификат. Да кажем, че търсим отговор на въпроса:

Вярно ли е, че уравнението $a_0 + a_1x + \cdots + a_nx^n = 0$ *има реален корен?*

По-лесно ще бъде да верифицираме някое предложено решение, т.е. да отговорим на въпроса:

Вярно ли е, че реалното число x_0 *е корен на уравнението* $a_0 + a_1x + \cdots + a_nx^n = 0$*?*

В него имаме още един параметър – предложеното решение x_0. В този случай това ще бъде сертификатът. Въпреки че в примера е направено така, не е нужно сертификатът да се използва. Той е само помощен и съществуването му е нужно само при отговор "ДА". Ясно е, че при отговор "НЕ" няма и как да има сертификат.
Неформално казано, в класа **P** се намират задачите, които се решават *"бързо"* (за полиномиално време), а в класа **NP** се намират задачите, чиито решения се верифицират *"бързо"*. Лесно може да се види, че $\mathbf{P} \subseteq \mathbf{NP}$. Ако една задача може да се реши за полиномиално време без да използва сертификат, то тогава тя може да се реши и със използване на сертификат.

8.2 Няколко важни задачи за класа NP

Следните задачи за изключително важни за класа **NP** (по-късно ще разберем защо):

- Задачата **SAT**:

  Вход: Съждителна формула φ в конюнктивна нормална форма.

  Въпрос: Има ли оценка, в която φ е вярна?

- Задачата **3SAT**:

  Вход: Съждителна формула φ в конюнктивна нормална форма, при която във всяка дизюнктивна клауза участват точно три литерала.

  Въпрос: Има ли оценка, в която φ е вярна?

- Задачата **SubsetSum**:

  Вход: Масив $A[1 \ldots n]$ от положителни числа и естествено число s.

  Въпрос: Има ли $I \subseteq \{1, \ldots, n\}$, за което $\sum_{i \in I} A[i] = s$?

- Задачата **2Partition**:

  Вход: Масив $A[1 \dots n]$ от положителни числа.

  Въпрос: Има ли $I \subseteq \{1, \dots, n\}$, за което $\sum_{i \in I} A[i] = \sum_{i \in \{1,\dots,n\} \setminus I} A[i]$?

- Задачата **VertexCover**:

  Вход: Граф $G = \langle V, E \rangle$ и естествено число k.

  Въпрос: Има ли $X \subseteq V$, за което $|X| \leq k$ и X съдържа поне един край на всяко ребро от E?

- Задачата **DominatingSet**:

  Вход: Граф $G = \langle V, E \rangle$ и естествено число k.

  Въпрос: Има ли $X \subseteq V$, за което $|X| \leq k$ и всеки връх от $V \setminus X$ е съседен на някой от X?

- Задачата **Clique**:

  Вход: Граф $G = \langle V, E \rangle$ и естествено число k.

  Въпрос: Има ли клика $X \subseteq V$, за която $|X| \geq k$?

- Задачата **Anticlique**:

  Вход: Граф $G = \langle V, E \rangle$ и естествено число k.

  Въпрос: Има ли антиклика $X \subseteq V$, за която $|X| \geq k$?

- Задачата **HamiltonianPath**:

  Вход: Граф $G = \langle V, E \rangle$ и два върха $s, e \in V$.

  Въпрос: Има ли Хамилтонов път в G от s до e?

- Задачата **SubgraphIsomorphism**:

  Вход: Два графа $G_1 = \langle V_1, E_1 \rangle$ и $G_2 = \langle V_2, E_2 \rangle$.

  Въпрос: Има ли подграф G на G_1, който е изоморфен на G_2?

- Задачата **TSP**:

  Вход: Тегловен граф $G = \langle V, E, w \rangle$ и естествено число k.

  Въпрос: Има ли Хамилтонов път в G с тегло ненадвишаващо k?

Нека покажем за някои от тях, че попадат в класа **NP**.
Да започнем със **SAT**. Трябва да предложим полиномиален алгоритъм, който с помощта на сертификат проверява за отговор "ДА". Сертификатът ще бъде оценката. Оценката може да се представи като масив от променливи $V[1 \dots k]$, чиито членове са променливите, които имат стойност "истина". Ясно е, че тази кодировка е с полиномиална относно формулата дължина. При дадена оценка лесно можем да видим дали формулата е вярна:

```
1  SAT(φ - съждителна формула в КНФ; V[1...k] - сертификат):
2      за всеки дизюнкт D във φ:
3          ако има литерал L в D, за който
4              (L = x и x ∈ V[1...k]) или (L = x̄ и x ∉ V[1...k]):
5              продължи нататък
6          иначе:
7              върни False
8
9      върни True
```

Този алгоритъм очевидно работи за полиномиално време. Наистина, той проверява дали $V[1 \dots k]$ задава оценка, в която φ е вярна. Така **SAT** е в класа **NP**. Тъй като **3SAT** е просто по-лесна версия на **SAT**, тя също попада в класа **NP**. Нека сега видим, че **SubsetSum** е в класа **NP**. Тук сертификатът ще бъде масив $I[1 \dots k]$, който ще представя множество от индекси за входния масив. При дадено множество от индекси лесно можем да проверим дали е изпълнено условието за сумата:

```
1  SubsetSum(A[1...n] - масив от положителни числа;
2            s - естествено число; I[1...k] - сертификат):
3      инициализирай S с 0
4
5      за всяко i от 1 до k:
6          прибави A[I[i]] към S
7
8      върни дали S = s?
```

Този алгоритъм очевидно работи за полиномиално време. Наистина, той проверява дали $I[1 \dots k]$ задава подмножество I на $\{1, \dots, n\}$, за което $\sum_{i \in I} A[i] = s$. Така **SubsetSum** е в класа **NP**.

Да видим сега, че **VertexCover** е в класа **NP**. Тук сертификатът ще бъде масив $X[1 \dots n]$, който ще представя множеството от върховете, които ще образуват потенциално върхово покритие. При дадено множество от върхове лесно можем да видим дали то е върхово покритие с размер най-много k:

```
1  VertexCover(G = (V, E) - граф;
2              k - естествено число; X[1...n] - сертификат):
3      ако n > k:
4          върни False
5
6      за всяко ребро (u, v) ∈ E:
7          ако u ∉ X[1...n] и v ∉ X[1...n]:
8              върни False
9
10     върни True
```

Този алгоритъм очевидно работи за полиномиално време. Наистина, той проверява дали $X[1 \dots n]$ задава върхово покритие на G с най-много k елемента. Така **VertexCover** е в класа **NP**.

Сега ще покажем, че **Clique** е в класа **NP**. Тук сертификатът ще бъде масив $X[1 \dots n]$, който ще представя множеството от върховете, които ще образуват потенциална клика. При дадено множество от върхове лесно можем да видим дали то е клика с размер поне k:

```
1  Clique(G = (V, E) - граф;
2         k - естествено число; X[1...n] - сертификат):
3      ако n < k:
4          върни False
5
6      за всеки връх u ∈ X[1...n]:
7          за всеки връх v ∈ X[1...n]:
8              ако u ≠ v и (u, v) ∉ E:
9                  върни False
10
11     върни True
```

Този алгоритъм очевидно работи за полиномиално време. Наистина, той проверява дали $X[1 \dots n]$ задава клика в G с поне k елемента. Така **Clique** е в класа **NP**.

8.3 P = NP или P ≠ NP? Колко пък да е трудно?

За съжаление, все още няма отговор на този въпрос – не се знае дали **P** = **NP** или **P** ≠ **NP**. Най-доброто, което имаме до момента, са няколко задачи в класа **NP**, които са изключително важни. Тези задачи в някакъв смисъл характеризират целия клас. Тях ще ги наричаме **NP**-пълни. Формалната дефиниция е следната, една изчислителна задача **X** е **NP**-пълна, ако:

- **X** е в класа **NP**;
- всяка задача **Y** в класа **NP** може алгоритмично да се сведе до задачата **X** за полиномиално време.

Фактът, че **Y** се свежда до **X** за полиномиално време, ще бележим с $\mathbf{Y} \leq_p \mathbf{X}$*. Лесно се вижда, че $\leq_p$ е транзитивна. Когато второто условие е изпълнено (без да е непременно изпълнено първото) казваме, че задачата **X** е **NP**-трудна.
Интересното за **NP**-пълните задачи е следното:

- ако покажем, че която и да е **NP**-пълна задача се решава за полиномиално време, то тогава **P** = **NP**;
- ако покажем, че която и да е **NP**-пълна задача не може да се реши за полиномиално време, то тогава **P** ≠ **NP**.

Всички **NP**-пълни задачи са еквивалентни в смисъл, че всяка може да се сведе до другата за полиномиално време. Така, имайки полиномиално алгоритъм, който решава някоя **NP**-пълна задача **X**, то тогава можем да решим всяка задача **Y** от **NP** за полиномиално време:

```
1  РешиY(ВходY):
2      ВходX ← РедуцирайВходYДоВходXЗаПолиномиалноВреме(ВходY)
3      ИзходX ← РешиXЗаПолиномиалноВреме(ВходX)
4      ИзходY ← РедуцирайИзходXДоИзходYЗаПолиномиалноВреме(ИзходX)
5      върни ИзходY
```

Обаче ние нито имаме такъв алгоритъм, нито не знаем, че такъв алгоритъм няма. Това са задачи, които хем не можем да решим ефективно, хем не можем да покажем, че такова решение няма. Най-доброто, което можем да направим, е да покажем, че задачата е еквивалентна по трудност на други задачи, за които други много умни хора не са се сетили.

*На други места вместо $\leq_p$ се използват означенията $\leq_m^p$ и $\propto_p$.

8.4 *"Основната"* NP-пълна задача

По-принцип е трудно директно да се показва **NP**-пълнота, ако се кара по дефиницията. Това, което обикновено се прави, е да се показва по дефиниция, че една задача е **NP**-пълна (все трябва да започнем отнякъде), след което се правят полиномиални редукции от задачи, за които знаем, че са **NP**-пълни, към задачите, които искаме да покажем, че са **NP**-пълни. Тук се възползваме от транзитивността на $\leq_p$. Разбира се, това само би показало **NP**-трудност, трябва и да се провери принадлежност към класа **NP**. Задачата, от която обикновено се започва е тази за удовлетворимост/изпълнимост.

Теорема 8.4.1 (Кук-Левин)**.** *Задачата **SAT** е **NP**-пълна.*

След това се показва, че **SAT** $\leq_p$ **3SAT**. Ние вече знаем, че тя е в класа **NP**, така че ако направим тази редукция, ще излезе, че **3SAT** е **NP**-пълна задача. Можем да направим следната редукция – за всеки дизюнкт D във входната формула φ:

- ако в D участва точно един литерал L, то тогава избираме нови променливи x и y и заменяме D с конюнкцията на дизюнктите $(L \vee x \vee y)$, $(L \vee x \vee \overline{y}), (L \vee \overline{x} \vee y)$ и $(L \vee \overline{x} \vee \overline{y})$;
- ако в D участват два литерала L_1, L_2, то тогава избираме нова променлива x и заменяме D с конюнкцията на дизюнктите $(L_1 \vee L_2 \vee x)$ и $(L_1 \vee L_2 \vee \overline{x})$;
- ако в D участват три литерала, не променяме D;
- ако в D участват литералите $L_1, \ldots, L_n$, където $n > 3$, то тогава избираме нови променливи $x_1, \ldots, x_{n-3}$ и заменяме D с конюнкцията на дизюнктите

  $(L_1 \vee L_2 \vee x_1), (L_3 \vee \overline{x_1} \vee x_2), (L_4 \vee \overline{x_2} \vee x_3), \ldots, (L_{n-2} \vee \overline{x_{n-4}} \vee x_{n-3}), (L_{n-1} \vee L_n \vee \overline{x_{n-3}})$.

Накрая ще получим като резултат формула ψ, която не е еквивалентна на φ, но е изпълнима т.с.т.к. φ е изпълнима. На читателя оставяме да провери, че това е вярно. Остана само да проверим, че всичко това става за полиномиално време. Дължината на ψ ще бъде полиномиална спрямо тази на φ, защото:

- дизюнктите с един литерал се заменят с четири дизюнкта с три литерала;
- дизюнктите с два литерала се заменят с два дизюнкта с три литерала;
- дизюнктите с три литерала не се променят;
- дизюнктите с $n > 4$ литерала се заменят с $n - 2$ дизюнкта с три литерала.

По-съмнителното е търсенето на нови променливи, но и това няма как да е прекалено бавно, защото броят на променливите, които участват в една формула е по-малък или равен на дължината ѝ. Тъй като ние ще получим формула с полиномиална на φ дължина, в нея ще участват най-много полиномиално на φ променливи. Това означава, че ако всеки път търсим неизползваната променлива с най-малък индекс, няма да търсим прекалено дълго. С това получаваме, че **3SAT** е **NP**-трудна задача, и понеже сме показвали че е в класа **NP**, то тя е и **NP**-пълна.

8.5 Класически NP-пълни задачи

Ще покажем няколко класически примера за **NP**-пълни задачи. Тъй като за тях сме доказвали, че са в класа **NP**, ни остава само да покажем, че са **NP**-трудни. Нека сега покажем, че **3SAT** $\leq_p$ **Clique**. Нека φ е формула в 3КНФ със n на брой дизюнкта. За полиномиално време ще построим граф G, за които φ е изпълнима т.с.т.к. G съдържа n-клика. За всеки дизюнкт $(L_1(x) \vee L_2(y) \vee L_3(z))$, който участва във φ, в графа G има върховете от вида $\{\langle x, v_x \rangle, \langle y, v_y \rangle, \langle z, v_z \rangle\}$, където $v_x, v_y, v_z \in \{\mathbb{T}, \mathbb{F}\}$, и интерпретирайки t като v_t за $t \in \{x, y, z\}$, дизюнктът $(L_1(x) \vee L_2(y) \vee L_3(z))$ се оценява като $\mathbb{T}$. Ребра ще има между тези множества, които не си противоречат, т.е. няма променлива x, за която $\langle x, \mathbb{T} \rangle$ да участва в едното множество и $\langle x, \mathbb{F} \rangle$ да участва в другото. По построение е очевидно, че в G има n клика т.с.т.к. φ е изпълнима. В едната посока кликата ни казва точно как да оценим променливите, а в другата посока от оценката можем да извлечем кликата. Тъй като за всеки дизюнкт получаваме най-много 2^3 върхове, то тогава конструкцията е полиномиална. С това показахме, че задачата **Clique** е **NP**-трудна, откъдето е и **NP**-пълна.
Сега нека да видим, че **Clique** $\leq_p$ **VertexCover**. За входния граф $G = \langle V, E \rangle$ строим $\overline{G} = \langle V, \overline{E} \rangle$, където:

$$\overline{E} = \{(u, v) \mid u, v \in V \ \& \ (u, v) \notin E\}.$$

Този граф очевидно се строи за време $\Theta(|V|^2)$.
Оказва се, че за всяко $X \subseteq V$ е изпълнено, че:

$$X \text{ е клика в } G \iff V \setminus X \text{ е върхово покритие в } \overline{G}.$$

($\Rightarrow$) Нека X е клика в G. Тогава което и ребро $(u, v) \in \overline{E}$ да вземем, $u \notin X$ или $v \notin X$. Ако $u, v \in X$, то тогава тъй като $(u, v) \in \overline{E}$, то тогава $(u, v) \notin E$, което противоречи с факта, че X е клика. Така наистина $V \setminus X$ е върхово покритие в $\overline{G}$.
($\Leftarrow$) Нека X не е клика в G. Тогава има два върха $u, v \in X$, за които $(u, v) \notin E$. Но тогава $(u, v) \in \overline{E}$, откъдето $V \setminus X$ не е върхово покритие в $\overline{G}$.
Псевдокодът на редукцията би изглеждал така:

```
1  РешиCliqueЧрезVertexCover(G = (V, E) граф; k - естествено число):
2      инициализирай граф Ḡ = (V, ∅)
3
4      за всеки връх u ∈ V:
5          за всеки връх v ∈ V:
6              ако u ≠ v и (u, v) ∉ E:
7                  добави реброто (u, v) към Ḡ
8
9      върни РешиVertexCover(Ḡ, |V| − k)
```

С това показахме, че **VertexCover** е **NP**-трудна задача, откъдето е и **NP**-пълна. Сега ще покажем, че **3SAT** $\leq_p$ **SubsetSum**. Нека е дадена формула φ в 3КНФ, в която участват променливите $x_1, \ldots, x_m$ и дизюнктите $D_1, \ldots, D_l$. Ще построим масив $A[1 \ldots n]$ и число k, за които:

$$\varphi \text{ е изпълнима } \iff \text{ има } S \subseteq \{1, \ldots, n\}, \text{ за което } k = \sum_{t \in S} A[t].$$

За всяка променлива x_i строим числа t_i, f_i със $m + l$ цифри по следния начин:

- i-тата цифра на t_i и f_i е 1;
- за $m + 1 \leq j \leq m + l$, j-тата цифра на t_i е 1, ако x_i участва в D_{j-m};
- за $m + 1 \leq j \leq m + l$, j-тата цифра на f_i е 1, ако $\overline{x_i}$ участва в D_{j-m};
- всички други цифри на t_i и f_i са 0.

Очевидно всички тези числа могат да се построят за полиномиално на дължината на φ време.
Сега за всеки дизюнкт D_j строим числа h_j и g_j със $m + l$ цифри по следния начин:

- $(m + j)$-тата цифра на h_j и g_j е 1;
- всички други цифри на h_j и g_j са 0.

Очевидно всички тези числа могат да се построят за полиномиално на дължината на φ време.
Накрая нека $k = \underbrace{1\ldots1}_{m \text{ пъти}}\underbrace{3\ldots3}_{l \text{ пъти}}$. Това число също се строи за полиномиално на дължината на φ време.
Накрая получаваме вход $[t_1, f_1, \ldots t_m, f_m, h_1, g_1, \ldots, h_l, g_l]$ и k за **SubsetSum**.
Нека φ е изпълнима т.е. е вярна при оценка v. Тогава за $1 \leq i \leq m$ избираме t_i, ако $v(x_i) = \mathbb{T}$, иначе избираме f_i. Тъй като тези числа не се бият в първите m цифри, те ще бъдат точно като в k, стига да не сумираме прекалено големи цифри от дясната част. За останалите l цифри на k можем да забележим, че в една формула ще са верни между 1 и 3 променливи. Тогава сумирайки избраните числа, за всяко $m+1 \leq j \leq m+l$ знаем, че j-тата цифра ще е между 1 и 3. Така можем просто да изберем достатъчно числа от h_j и g_j, че да добутаме до 3.
Обратно, нека имаме $I \subseteq \{1, \ldots, 2m + 2l\}$, за което $k = \sum_{i \in I} A[i]$. От структурата на числото k лесно се вижда, че за всяко $1 \leq i \leq m$ точно едно от t_i и f_i участва в $\{A[i] \mid i \in I\}$. Нека $v(x_i) = \mathbb{T}$ при $t_i \in S$ и $v(x_i) = \mathbb{F}$, иначе. Ако има такъв дизюнкт D_j, който при оценката v се остойностява като $\mathbb{F}$, то тогава $(m+j)$-тата цифра на k щеше да е строго по-малка от 3. Следователно всички дизюнкти във φ са верни при оценка v, откъдето φ е изпълнима.
Псевдокодът на редукцията би изглеждал така:

```
1  РешиЗSATЧрезSubsetSum(φ - формула в ЗКНФ с променливи x_1, ..., x_m
2                            и дизюнкти D_1, ..., D_l):
3      създай масив A[1...2m+2l] и инициализирай t с 1
4
5      за всяко 1 ≤ i ≤ m:
6          A[t] ← Постройt(m, l, x_i, D_1, ..., D_l)
7          A[t+1] ← Постройf(m, l, x_i, D_1, ..., D_l)
8          t ← t+2
9
10     за всяко 1 ≤ j ≤ l:
11         A[t] ← ПостройhИлиg(m, l)
12         A[t+1] ← ПостройhИлиg(m, l)
13         t ← t+2
14
15     k ← Постройk(m, l)
16     върни РешиSubsetSum(A[1...2m+2l], k)
```

С това показахме, че **SubsetSum** е **NP**-трудна задача, откъдето е и **NP**-пълна.

8.6 Задачи

Задача 8.1. Да се докаже, че следните задачи са в класа **NP**:

- **2Partition**;
- **DominatingSet**;
- **Anticlique**;
- **HamiltonianPath**;
- **SubgraphIsomorphism**;
- **TSP**.

Задача 8.2. Разглеждаме задачата **StarFreeRegexIneq**:

Вход: Два регулярни израза r_1 и r_2, в които не участва $*$.
Въпрос: Вярно ли е, че $\mathcal{L}[\![r_1]\!] \neq \mathcal{L}[\![r_2]\!]$?

Да се докаже, че задачата **StarFreeRegexIneq** е в класа **NP**.

Задача 8.3. Разглеждаме задачата **ChromaticNumber**:

Вход: Граф $G = \langle V, E \rangle$ и естествено число k.
Въпрос: Вярно ли е, че хроматичното число на G не надвишава k?

Да се докаже, че задачата **ChromaticNumber** е в класа **NP**.

Задача 8.4. Да се докаже, че **Anticlique** е **NP**-пълна задача.

Задача 8.5. Да се докаже, че **2Partition** е **NP**-пълна задача.

Задача 8.6. Да се докаже, че **DominatingSet** е **NP**-пълна задача.